

z/OS



# XL C/C++ Compiler and Runtime Migration Guide for the Application Programmer

*Version 2 Release 1*

**Note**

Before using this information and the product it supports, read the information in "Notices" on page 149.

This edition applies to Version 2 Release 1 of z/OS (5650-ZOS) and to all subsequent releases and modifications until otherwise indicated in new editions.

© **Copyright IBM Corporation 2015.**

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

---

# Contents

<b>About this document . . . . .</b>	<b>ix</b>
z/OS XL C/C++ on the World Wide Web . . . . .	xvi
Where to find more information . . . . .	xvi
Technical support . . . . .	xvi
How to send your comments to IBM . . . . .	xvii
If you have a technical problem . . . . .	xviii

---

## Part 1. Introduction . . . . . 1

<b>Chapter 1. New migration issues for z/OS XL C/C+ V2R1M1 . . . . .</b>	<b>3</b>
--	----------

<b>Chapter 2. New migration issues for z/OS V2R1 XL C/C++ . . . . .</b>	<b>5</b>
---	----------

### Chapter 3. Program migration checklists 9

Before you start your migration . . . . .	9
When you are compiling code . . . . .	10
When you are binding program objects or load modules . . . . .	11
When you are running an application . . . . .	12
Tools that facilitate your migration . . . . .	13
The Edge Portfolio Analyzer . . . . .	13
Applicability of product information . . . . .	14
Version history of IBM C/C++ compilers and libraries . . . . .	15

---

## Part 2. Migration of pre-OS/390 C/C++ applications to z/OS V2R1 XL C/C++ . . . . . 17

### Chapter 4. Source code compatibility issues with pre-OS/390 C/C++ programs . . . . . 19

Removal of IBM Open Class Library support . . . . .	19
Source code modifications necessitated by changes in runtime library . . . . .	19
The #pragma runopts directive . . . . .	19
Resource allocation and memory management issues . . . . .	19
The sizeof operator applied to a function return type . . . . .	20
A user-defined global new operator and array new . . . . .	20
Addressing incompatibilities . . . . .	20
C/370 V2 main program and main entry point . . . . .	21
Pointer incompatibilities . . . . .	21
Data type incompatibilities . . . . .	21
Assignment restrictions for packed structures and unions . . . . .	21
DSECT header files and packed structures . . . . .	21

Changes required by programs with interlanguage calls . . . . .	22
Explicit program mask manipulations . . . . .	22
Assembler source code changes in System Programming C (SPC) applications built with EDCXSTRX . . . . .	22
Internationalization incompatibilities . . . . .	23
Support of alternate code points . . . . .	23

### Chapter 5. Compile-time issues with pre-OS/390 C/C++ programs . . . . . 25

Changes in compiler listings, messages, and return codes . . . . .	25
Macro redefinitions might result in severe errors . . . . .	25
Changes in compiler options . . . . .	25
Compiler options that are no longer supported . . . . .	25
Compiler options that were introduced in OS/390 C/C++ or later . . . . .	26
Changes in compiler option functionality . . . . .	26
Changes that affect compiler invocations . . . . .	29
IPA compiler option and very large applications . . . . .	29
Customized JCL and the CXX format . . . . .	29
CBCI and CBCXI procedures in JCL . . . . .	30
Changes that affect SYSLIB DD cards . . . . .	30
Change in SCLBH logical record length . . . . .	30

### Chapter 6. Bind-time migration issues with pre-OS/390 C/C++ programs . . . . . 31

Library release level in use . . . . .	31
Binder invocation changes . . . . .	33
Impact of changes to CC EXEC invocation syntax . . . . .	33
Changes due to customizations of the runtime environment . . . . .	33
User-developed exit routines . . . . .	33
Incompatibilities in external references . . . . .	34
Requirements for relinking C/370 modules that invoke Debug Tool . . . . .	34
C/370 modules with interlanguage calls (ILC) . . . . .	34
Interlanguage calls between assembler and PL/I language modules . . . . .	35
Function calls between C and Fortran modules . . . . .	35
Function calls to and from COBOL modules . . . . .	35

### Chapter 7. Runtime migration issues with pre-OS/390 C/C++ applications . . . . . 41

Retention of pre-OS/390 runtime behavior . . . . .	41
Runtime library messages . . . . .	41
Return codes and messages . . . . .	41
Error conditions that cause runtime messages . . . . .	42
Prefixes of perror() and strerror() messages . . . . .	42
Language specification for messages . . . . .	42
User-developed exit routines . . . . .	42
Changes that affect customized JCL procedures . . . . .	43
Changes in data set names . . . . .	43

Arguments that contain a slash . . . . .	43
Differences in standard streams . . . . .	43
Dump generation . . . . .	43
Changes in runtime option specification . . . . .	43
Runtime options lists . . . . .	43
Obsolete runtime options . . . . .	43
Return codes for abnormal enclave terminations	44
Abnormal terminations and the TRAP runtime option . . . . .	44
Default heap allocations . . . . .	44
HEAP parameter specification . . . . .	44
Default stack allocations . . . . .	45
STACK parameter specification . . . . .	45
XPLINK downward-growing stack and the THREADSTACK runtime option . . . . .	45
Runtime library compatibility issues with pre-OS/390 applications . . . . .	45
Changes to the putenv() function and POSIX compliance . . . . .	45
UCMAPS and UCS-2 and UTF-8 converters . . . . .	46
Common library initialization compatibility issues with C/370 modules . . . . .	46
Internationalization issues in POSIX and non-POSIX applications . . . . .	47
Hardware and OS exceptions . . . . .	48
Decimal overflow exceptions . . . . .	48
SIGTERM, SIGINT, SIGUSR1, and SIGUSR2 exceptions . . . . .	48
Unexpected SIGFPE exceptions . . . . .	48
Resource allocation and memory management migration issues . . . . .	49
The realloc() function . . . . .	49

**Chapter 8. Input and output operations compatibility . . . . . 51**

Migration issues when opening pre-OS/390 files . . . . .	51
Migration issues when writing to pre-OS/390 files . . . . .	51
Changes in DBCS string behavior . . . . .	53
Changes in stdout and stderr file positioning . . . . .	53
Behavior changes when closing and reopening ASA files . . . . .	55
Changes in values returned by the fldata() function . . . . .	55
VSAM I/O changes . . . . .	55
Change in allocation of VSAM control blocks and I/O buffers . . . . .	56
Terminal I/O changes . . . . .	56

**Part 3. Migration of OS/390 C/C++ applications to z/OS V2R1 XL C/C++ 57**

**Chapter 9. Source code compatibility issues with OS/390 programs . . . . . 59**

Overflow processing and code modifications . . . . .	59
References to class libraries that are no longer shipped . . . . .	59

**Chapter 10. Compile-time migration issues with OS/390 programs . . . . . 61**

Changes in compiler listings and messages . . . . .	61
---	----

Debug format specification . . . . .	61
Language specification for compiler messages . . . . .	61
Optimization level mapping and listing content . . . . .	62
Macro redefinitions and error messages . . . . .	62
Changes in compiler options . . . . .	62
Compiler options that are no longer supported . . . . .	62
ARCHITECTURE compiler option . . . . .	63
ARGPARSE compiler option with Metal . . . . .	63
ASCII compiler option . . . . .	63
CHECKOUT(CAST) compiler option . . . . .	64
DIGRAPH compiler option . . . . .	64
ENUMSIZE compiler option . . . . .	64
INFO compiler option . . . . .	64
INLINE compiler option . . . . .	64
IPA(LINK) compiler option . . . . .	65
LANGLVL(ANSI), LANGLVL(SAA), or LANGLVL(SAAL2) compiler option and macro redefinitions . . . . .	66
LANGLVL(EXTENDED) compiler option and macro redefinitions . . . . .	66
LANGLVL(LONGLONG) compiler option . . . . .	67
LOCALE compiler option . . . . .	67
M compiler option . . . . .	67
OPTIMIZE compiler option . . . . .	68
NORENT compiler option . . . . .	68
ROSTRING compiler option . . . . .	68
ROCONST compiler option . . . . .	68
STATICINLINE compiler option . . . . .	68
SQL compiler option and SQL EXEC statements . . . . .	68
TARGET compiler option . . . . .	69
TEST compiler option . . . . .	69
TUNE compiler option . . . . .	69
Changes in IBM data set names . . . . .	69
Introduction of 1998 Standard C++ support . . . . .	69
Changes that affect performance and optimization . . . . .	69
Addition of the #pragma reachable and #pragma leaves directives . . . . .	69
Changes that affect customized JCL procedures . . . . .	70
Potential increase in memory requirements . . . . .	70
JCL CBCI and CBCXI procedures and the variable CLBPRFX . . . . .	70
Syntax to invoke the CC command . . . . .	70
Removal of Model Tool support . . . . .	70

**Chapter 11. Bind-time migration issues with OS/390 C/C++ programs . . . . . 71**

Reentrant variables when the compiler option is NORENT . . . . .	71
--	----

**Chapter 12. Runtime migration issues with OS/390 C/C++ applications . . . . . 73**

Retention of OS/390 runtime behavior . . . . .	73
Changes to the putenv() function and POSIX compliance . . . . .	73
Debug format and translation of the c89 -g flag option . . . . .	74
Language Environment customization issues . . . . .	74
Change in allocation of VSAM control blocks . . . . .	74

## Chapter 13. Migration issues resulting from class library changes between OS/390 C/C++ applications and Standard C++ library. . . . . 75

Function calls to different libraries. . . . .	75
Removal of IBM Open Class Library support . . . . .	75
Removal of SOM support. . . . .	75
Removal of Database Access Class Library utility. . . . .	75
Migration of programs with calls to UNIX System Laboratories I/O Stream Library functions . . . . .	75

## Part 4. Migration of earlier z/OS C/C++ applications to z/OS V2R1 XL C/C++ . . . . . 77

### Chapter 14. Source code compatibility issues with earlier z/OS C/C++ programs . . . . . 79

Function calls to different libraries. . . . .	79
References to class libraries that are no longer shipped . . . . .	79
Migration from UNIX System Laboratories I/O Stream Library to Standard C++ I/O Stream Library	80
Standard C++ compliance compatibility issues. . . . .	80
Use of XL C/C++ library functions . . . . .	80
Timing of processor release by the pthread_yield() function . . . . .	80
New information returned by the getnameinfo() function . . . . .	81
Feature test macros and system header files . . . . .	81
Potential need to include _Ieee754.h . . . . .	81
New definitions exposed by use of the _OPEN_SYS_SOCKET_IPV6 macro . . . . .	82
Required changes to fprintf and fscanf strings %D, %DD, and %H. . . . .	82
Changes to the putenv() function and POSIX compliance . . . . .	82
C99 support of long long data type . . . . .	83
Use of pragmas . . . . .	83
Application of #pragma unroll() as of z/OS V1R7 XL C/C++. . . . .	83
Unexpected C++ output with #pragma pack(2) . . . . .	84
Virtual function declaration and use . . . . .	84

### Chapter 15. Compile-time migration issues with earlier z/OS C/C++ programs . . . . . 87

Changes in compiler listings, messages, and return codes . . . . .	87
Appearance of compiler substitution variables. . . . .	88
Corrections in escape sequence encoding . . . . .	88
Function offsets in source listing . . . . .	88
Diagnostic refinement in identification of linkage issues (C++ only) . . . . .	88
References to UNIX System Services file names . . . . .	89
Non-compliant array index raises an exception . . . . .	89
Unexpected name lookup error messages with template use . . . . .	90

Width of mnemonic in assembly listings. . . . .	90
Macro redefinitions and error messages . . . . .	90
Changes in compiler option functionality . . . . .	91
Option behavior change when processing multiple suboptions . . . . .	91
CHECKOUT compiler option . . . . .	91
CMDOPTS compiler option and conflict resolution . . . . .	91
DFP compiler option and earlier floating-point applications . . . . .	91
DSAUSER compiler option . . . . .	92
ENUMSIZE(SMALL) and protected enumeration types in system header files . . . . .	92
FLAG compiler option. . . . .	93
FLOAT(AFP) suboptions for applications that access CICS data . . . . .	93
GENASM compiler option . . . . .	93
GONUMBER compiler option and LP64 support . . . . .	93
IPA compiler option . . . . .	93
LANGLVL(ANSI), LANGLVL(SAA), or LANGLVL(SAAL2) compiler option and macro redefinitions . . . . .	93
LANGLVL(EXTC1X) compiler option. . . . .	93
LANGLVL(EXTENDED) compiler option and macro redefinitions . . . . .	94
LANGLVL(EXTENDED0X) compiler option . . . . .	94
LOCALE compiler option . . . . .	94
M compiler option . . . . .	95
RESTRICT option . . . . .	95
SEVERITY option . . . . .	95
SQL compiler option and SQL EXEC statements . . . . .	95
TARGET compiler option. . . . .	95
TEMPLATEDEPTH compiler option . . . . .	95
Changes that affect compiler invocations . . . . .	95
Changes that affect use of the c89 command . . . . .	96
Changes that affect use of the xlc utility . . . . .	97
Changes that affect JCL procedures . . . . .	97
User-defined conversion tables and iconv() functions . . . . .	98
ILP32 compiler option and name mangling. . . . .	98
IPA(LINK) compiler option and very large applications . . . . .	98
IPA(LINK) compiler option and exploitation of 64-bit virtual memory . . . . .	99
JCL that runs pre-z/OS V1R5 C/C++ programs . . . . .	99
Compiler options that manage Standard C++ compliance . . . . .	99
Impact of recompiling applications that include <net/if.h> with the _XOPEN_SOURCE_EXTENDED feature test macro. . . . .	100
Impact of recompiling applications that include the pselect() interface . . . . .	100
Impact of recompiling with the _OPEN_SYS_SOCKET_IPV6 macro . . . . .	100
Impact of recompiling code that relies on math.h to include IEEE 754 interfaces. . . . .	100

**Chapter 16. Bind-time migration issues with earlier z/OS C/C++ programs . . . . . 101**

Unexpected "missing symbol" error (C++ only) . . . 101  
Program modules from an earlier release . . . 101  
  Namespace pollution binder errors . . . . 101  
  c89 COMPAT binder option default and programs from an earlier release . . . . . 102  
Alignment incompatibilities between object models 102  
  Alignment incompatibilities between XL C and XL C++ output with #pragma pack(2) . . . . 102  
Debug format and c89 -g flag option translation 103  
argc argv parsing support for Metal C programs 103

**Chapter 17. Runtime migration issues with earlier z/OS C/C++ applications . 105**

Earlier AMODE 64 applications . . . . . 105  
  HEAPOOLS runtime option no longer ignored in all AMODE 64 applications . . . . . 105  
Customized runtime libraries . . . . . 105  
Failure of authentication process . . . . . 106  
Retention of previous runtime behavior . . . . 106  
  Unexpected output from fprintf() or fscanf() . . . 106  
  IEEE754 math functions . . . . . 107  
  Internal timing algorithm specification . . . . 107  
  Daylight saving time definition . . . . . 107  
  Changes to the putenv() function and POSIX compliance . . . . . 107  
Internationalization issues . . . . . 108  
  Default daylight saving time change. . . . . 108  
  EEC default currency update . . . . . 108  
  Movement of LOCALDEF utilities to new data sets. . . . . 108  
Changes in math library functions . . . . . 109  
Changes in floating-point support . . . . . 110  
  Hexadecimal floating-point notation . . . . . 110  
  Floating-point special values . . . . . 111  
Changes in allocation of VSAM control blocks . . 111  
Changes to st\_mode attribute of AF\_UNIX socket files. . . . . 111  
Changes to strfmon() output . . . . . 112  
Changes to structure t\_opthdr in xti.h . . . . 112  
Changes to getting group or user database entry 112  
Removal of conversion table source code . . . . 112

---

**Part 5. ISO Standard C++ compliance migration issues . . . 115**

**Chapter 18. Language level and your Standard C++ compliance objectives . 117**

**Chapter 19. Changes that affect Standard C++ compliance of language features . . . . . 119**

Unqualified name lookups and the using directive 119  
Order of destruction for statically initialized objects 120  
Implicit integer type declarations . . . . . 120  
Scope of for-loop initializer declarations . . . . 120

Visibility of friend declarations . . . . . 121  
Migration of friend declarations in class member lists . . . . . 121  
cv-qualifications when the thrown and caught types are the same. . . . . 122  
Compiler options that are introduced in C++11 standard . . . . . 122  
  LANGLVL(AUTOTYPEDEDUCTION) compiler option (C++11) . . . . . 123  
  LANGLVL(C1XNORETURN) compiler option (C++11) . . . . . 123  
  LANGLVL(C99LONGLONG) compiler option (C++11) . . . . . 123  
  LANGLVL(C99PREPROCESSOR) compiler option (C++11) . . . . . 123  
  LANGLVL(CONSTEXPR) compiler option (C++11) . . . . . 124  
  LANGLVL(DECLTYPE) compiler option (C++11) 124  
  LANGLVL(DEFAULTANDDELETE) compiler option (C++11) . . . . . 124  
  LANGLVL(DELEGATINGCTORS) compiler option (C++11) . . . . . 124  
  LANGLVL(EXPLICITCONVERSIONOPERATORS) compiler option (C++11). . . . . 124  
  LANGLVL(EXTENDED FRIEND) compiler option (C++11) . . . . . 124  
  LANGLVL(EXTENDEDINTEGERSAFE) compiler option (C++11). . . . . 125  
  LANGLVL(EXTERNTEMPLATE) compiler option (C++11) . . . . . 125  
  LANGLVL(INLINENAMESPACE) compiler option (C++11) . . . . . 125  
  LANGLVL(REFERENCECOLLAPSING) compiler option (C++11). . . . . 125  
  LANGLVL(RIGHTANGLEBRACKET) compiler option (C++11) . . . . . 125  
  LANGLVL(RVALUEREFERENCES) compiler option (C++11) . . . . . 126  
  LANGLVL(SCOPEDENUM) compiler option (C++11) . . . . . 126  
  LANGLVL(STATIC\_ASSERT) compiler option (C++11) . . . . . 126  
  LANGLVL(VARIADICTEMPLATES) compiler option (C++11) . . . . . 126  
  WARN0X compiler option (C++11) . . . . . 126  
Errors due to changes in compiler behavior . . . 126  
  C++ class access errors . . . . . 126  
  Exceptions caused by ambiguous overloads . . . 127  
  Exceptions caused by user-defined conversions 128  
  Issues caused by the use of incomplete types in exception-specifications . . . . . 128  
  Syntax errors with array new . . . . . 129

---

**Part 6. Migration issues for C/C++ applications that use other IBM products . . . . . 131**

**Chapter 20. Migration issues with earlier C/C++ applications that run CICS statements . . . . . 133**

Migration of CICS statements from pre-OS/390 C/C++ applications . . . . . 133

- CICS statement translation options . . . . . 133
- HEAP option used with the interface to CICS . . . . . 133
- User-developed exit routines . . . . . 133
- Multiple libraries under CICS . . . . . 133
- CICS abend codes and messages . . . . . 134
- CICS reason codes. . . . . 134
- Standard stream support under CICS . . . . . 134
- Changes in stderr output under CICS . . . . . 135
- Transient data queue names under CICS . . . . . 135

Migration of CICS statements from earlier XL C/C++ applications . . . . . 135

- CICS TS V4.1 with "Extended MVS Linkage Convention". . . . . 136
- Customized CEECCSD.COPY and CEECCSDX.COPY files and iconv() changes . . . 136

**Chapter 21. Migration issues with earlier C/C++ applications that use DB2 Universal Database. . . . . 139**

Namespace violations and SQL coprocessor-based compilations. . . . . 139

Example: Performing a macro definition check . . . . . 140

Example: Explicitly undefining and redefining a macro . . . . . 140

Potential need to specify DBRMLIB with the SQL option. . . . . 141

---

**Part 7. Appendixes . . . . . 143**

**Appendix. Accessibility . . . . . 145**

Accessibility features . . . . . 145

Consult assistive technologies . . . . . 145

Keyboard navigation of the user interface . . . . . 145

Dotted decimal syntax diagrams . . . . . 145

**Notices . . . . . 149**

Policy for unsupported hardware. . . . . 150

Minimum supported hardware . . . . . 151

Programming interface information . . . . . 151

Standards . . . . . 151

Trademarks . . . . . 152

**Bibliography. . . . . 153**

**Index . . . . . 155**





---

## About this document

This document discusses the implications of migrating applications from each of the supported compilers and libraries to the IBM® z/OS® V2R1 XL C/C++ release. To find the section of the document that applies to your migration, see “How to use this document.”

This document contains terminology, maintenance, and editorial changes. Technical changes or additions to the text and illustrations are indicated by a vertical line (|) to the left of the change.

You may notice changes in the style and structure of some of the contents in this document; for example, headings that use uppercase for the first letter of initial words only, and procedures that have a different look and format. The changes are ongoing improvements to the consistency and retrievability of information in our documents.

### How to use this document

You can use this document to:

- Help determine whether and how you can continue to use existing source code, object code, and load modules
- Become aware of the changes in compiler and runtime behavior that may affect your migration from earlier versions of the compiler

**Note:** In most situations, existing well-written applications can continue to work without modification.

*This document does not:*

- Discuss all of the enhancements that have been made to the z/OS XL C/C++ compiler and IBM Language Environment® element provided with z/OS.

#### Notes:

1. All subsequent "Language Environment" references in this document apply to the Language Environment element that is provided with the z/OS operating system unless otherwise specified as applying to an earlier operating system.
  2. For a list of books that provide information about the z/OS XL C/C++ compiler and Language Environment element, refer to “z/OS XL C/C++ and related documents” on page xi.
- Show how to change an existing C program so that it can use C++.

**Note:** For a description of some of the differences between C and C++, see *z/OS XL C/C++ Language Reference*.

### How this document is organized

This document includes the following topics:

- Part 1 provides information that you will need to understand before you migrate programs or applications, as well as assistance in finding the information that is

relevant to your migration. See Chapter 1, “New migration issues for z/OS XL C/C+ V2R1M1,” on page 3 and Chapter 3, “Program migration checklists,” on page 9.

- Part 2 describes the considerations for migrating from a pre-OS/390 C and C++ application. See Part 2, “Migration of pre-OS/390 C/C++ applications to z/OS V2R1 XL C/C++,” on page 17.
- Part 3 describes the considerations for migrating from an IBM OS/390® C and C++ application. See Part 3, “Migration of OS/390 C/C++ applications to z/OS V2R1 XL C/C++,” on page 57.
- Part 4 describes the considerations for migrating from an earlier z/OS C/C++ application. See Part 4, “Migration of earlier z/OS C/C++ applications to z/OS V2R1 XL C/C++,” on page 77.
- Part 5 describes the migration issues related to *Programming languages - C++ (ISO/IEC 14882:2003(E))*, which documents the C++ Standard. See Part 5, “ISO Standard C++ compliance migration issues,” on page 115.
- Part 6 describes the issues related to migration of C/C++ programs that access IBM CICS® or IBM DB2® information. See Part 6, “Migration issues for C/C++ applications that use other IBM products,” on page 131.

Within Parts 2, 3, and 4, chapters are organized around the following areas:

- Possible changes to source code that are required by the migration.
- Migration issues that affect compilations.
- Migration issues that affect the linking or binding process.
- Migration issues that affect application execution.
- Migration issues that are caused by class library changes.

In this release of the document, you will notice that some topics are covered in different locations. Use the index to see all discussions related to a specific topic, such as POSIX compliance or globalization. The index is structured to support quick and selective retrieval of specific topics.

## Typographical conventions

The following table explains the typographical conventions used in this document.

Table 1. *Typographical conventions*

Typeface	Indicates	Example
<b>bold</b>	Commands, executable names, compiler options and pragma directives that contain lower-case letters.	The xlc utility provides two basic compiler invocation commands, <b>xlc</b> and <b>xlc (xlc++)</b> , along with several other compiler invocation commands to support various C/C++ language levels and compilation environments.
<i>italics</i>	Parameters or variables whose actual names or values are to be supplied by the user. Italics are also used to introduce new terms.	Make sure that you update the <i>size</i> parameter if you return more than the <i>size</i> requested.
monospace	Programming keywords and library functions, compiler built-in functions, file and directory names, examples of program code, command strings, or user-defined names.	If one or two cases of a <code>switch</code> statement are typically executed much more frequently than other cases, break out those cases by handling them separately before the <code>switch</code> statement.

## z/OS XL C/C++ and related documents

This topic summarizes the content of the z/OS XL C/C++ documents and shows where to find related information in other documents.

Table 2. z/OS XL C/C++ and related documents

Document Title and Number	Key Sections/Chapters in the Document
<i>z/OS XL C/C++ Programming Guide, SC14-7315</i>	<p>Guidance information for:</p> <ul style="list-style-type: none"><li>• XL C/C++ input and output</li><li>• Debugging z/OS XL C programs that use input/output</li><li>• Using linkage specifications in C++</li><li>• Combining C and assembler</li><li>• Creating and using DLLs</li><li>• Using threads in z/OS UNIX System Services applications</li><li>• Reentrancy</li><li>• Handling exceptions, error conditions, and signals</li><li>• Performance optimization</li><li>• Network communications under z/OS UNIX</li><li>• Interprocess communications using z/OS UNIX</li><li>• Structuring a program that uses C++ templates</li><li>• Using environment variables</li><li>• Using System Programming C facilities</li><li>• Library functions for the System Programming C facilities</li><li>• Using runtime user exits</li><li>• Using the z/OS XL C multitasking facility</li><li>• Using other IBM products with z/OS XL C/C++ (IBM CICS Transaction Server for z/OS, CSP, DWS, IBM DB2, IBM GDDM, IBM IMS™, ISPF, IBM QMF™)</li><li>• Globalization: locales and character sets, code set conversion utilities, mapping variant characters</li><li>• POSIX character set</li><li>• Code point mappings</li><li>• Locales supplied with z/OS XL C/C++</li><li>• Charmap files supplied with z/OS XL C/C++</li><li>• Examples of charmap and locale definition source files</li><li>• Converting code from coded character set IBM-1047</li><li>• Using built-in functions</li><li>• Using vector programming support</li><li>• Using runtime check library</li><li>• Using high performance libraries</li><li>• Programming considerations for z/OS UNIX C/C++</li></ul>

Table 2. z/OS XL C/C++ and related documents (continued)

Document Title and Number	Key Sections/Chapters in the Document
z/OS XL C/C++ User's Guide, SC14-7307	<p>Guidance information for:</p> <ul style="list-style-type: none"> <li>• z/OS XL C/C++ examples</li> <li>• Compiler options</li> <li>• Binder options and control statements</li> <li>• Specifying Language Environment runtime options</li> <li>• Compiling, IPA Linking, binding, and running z/OS XL C/C++ programs</li> <li>• Utilities (Object Library, CXXFILT, DSECT Conversion, Code Set and Locale, ar and make, BPXBATCH, c89, xlc)</li> <li>• Diagnosing problems</li> <li>• Cataloged procedures and IBM REXX EXECs</li> <li>• Customizing default options for the z/OS XL C/C++ compiler</li> </ul>
z/OS XL C/C++ Language Reference, SC14-7308	<p>Reference information for:</p> <ul style="list-style-type: none"> <li>• The C and C++ languages</li> <li>• Lexical elements of z/OS XL C and C++</li> <li>• Declarations, expressions, and operators</li> <li>• Implicit type conversions</li> <li>• Functions and statements</li> <li>• Preprocessor directives</li> <li>• C++ classes, class members, and friends</li> <li>• C++ overloading, special member functions, and inheritance</li> <li>• C++ templates and exception handling</li> <li>• z/OS XL C and C++ compatibility</li> </ul>
z/OS XL C/C++ Messages, GC14-7305	<p>Provides error messages and return codes for the compiler, and its related application interface libraries and utilities. For the XL C/C++ runtime library messages, refer to <i>z/OS Language Environment Runtime Messages, SA38-0686</i>. For the c89 and xlc utility messages, refer to <i>z/OS UNIX System Services Messages and Codes, SA23-2284</i>.</p>
z/OS XL C/C++ Runtime Library Reference, SC14-7314	<p>Reference information for:</p> <ul style="list-style-type: none"> <li>• header files</li> <li>• library functions</li> </ul>
z/OS C Curses, SA38-0690	<p>Reference information for:</p> <ul style="list-style-type: none"> <li>• Curses concepts</li> <li>• Key data types</li> <li>• General rules for characters, renditions, and window properties</li> <li>• General rules of operations and operating modes</li> <li>• Use of macros</li> <li>• Restrictions on block-mode terminals</li> <li>• Curses functional interface</li> <li>• Contents of headers</li> <li>• The terminfo database</li> </ul>

Table 2. z/OS XL C/C++ and related documents (continued)

Document Title and Number	Key Sections/Chapters in the Document
<i>z/OS XL C/C++ Compiler and Runtime Migration Guide for the Application Programmer, GC14-7306</i>	<p>Guidance and reference information for:</p> <ul style="list-style-type: none"> <li>• Common migration questions</li> <li>• Application executable program compatibility</li> <li>• Source program compatibility</li> <li>• Input and output operations compatibility</li> <li>• Class library migration considerations</li> <li>• Changes between releases of z/OS</li> <li>• Pre-z/OS C and C++ compilers to current compiler migration</li> <li>• Other migration considerations</li> </ul>
<i>z/OS Metal C Programming Guide and Reference, SC14-7313</i>	<p>Guidance and reference information for:</p> <ul style="list-style-type: none"> <li>• Metal C run time</li> <li>• Metal C programming</li> <li>• AR mode</li> </ul>
<i>Standard C++ Library Reference, SC14-7309</i>	<p>The documentation describes how to use the following three main components of the Standard C++ Library to write portable C/C++ code that complies with the ISO standards:</p> <ul style="list-style-type: none"> <li>• ISO Standard C Library</li> <li>• ISO Standard C++ Library</li> <li>• Standard Template Library (C++)</li> </ul> <p>The ISO Standard C++ library consists of 51 required headers. These 51 C++ library headers (along with the additional 18 Standard C headers) constitute a hosted implementation of the C++ library. Of these 51 headers, 13 constitute the Standard Template Library, or STL.</p>
<i>z/OS Common Debug Architecture User's Guide, SC14-7310</i>	<p>This documentation is the user's guide for IBM's libddpi library. It includes:</p> <ul style="list-style-type: none"> <li>• Overview of the architecture</li> <li>• Information on the order and purpose of API calls for model user applications and for accessing DWARF information</li> <li>• Information on using the Common Debug Architecture with C/C++ source</li> </ul> <p>This user's guide is part of the Runtime Library Extensions documentation.</p>
<i>z/OS Common Debug Architecture Library Reference, SC14-7311</i>	<p>This documentation is the reference for IBM's libddpi library. It includes:</p> <ul style="list-style-type: none"> <li>• General discussion of Common Debug Architecture</li> <li>• Description of APIs and data types related to stacks, processes, operating systems, machine state, storage, and formatting</li> </ul> <p>This reference is part of the Runtime Library Extensions documentation.</p>
<i>DWARF/ELF Extensions Library Reference, SC14-7312</i>	<p>This documentation is the reference for IBM's extensions to the libdwarf and libelf libraries. It includes information on:</p> <ul style="list-style-type: none"> <li>• Consumer APIs</li> <li>• Producer APIs</li> </ul> <p>This reference is part of the Runtime Library Extensions documentation.</p>
Debug Tool documentation, available on the Debug Tool for z/OS library page on the World Wide Web	<p>The documentation, which is available at <a href="http://www.ibm.com/software/awdtools/debugtool/library/">http://www.ibm.com/software/awdtools/debugtool/library/</a>, provides guidance and reference information for debugging programs, using Debug Tool in different environments, and language-specific information.</p>

Table 2. z/OS XL C/C++ and related documents (continued)

Document Title and Number	Key Sections/Chapters in the Document
README file	The README file provides additional information for using the z/OS XL C/C++ licensed program, including late changes to z/OS XL C/C++ publications. To access any README files that were published after the ship date, go to <a href="http://www.ibm.com/support/docview.wss?uid=swg27007531">http://www.ibm.com/support/docview.wss?uid=swg27007531</a> .

**Note:** For complete and detailed information on linking and running with Language Environment services and using the Language Environment runtime options, refer to *z/OS Language Environment Programming Guide, SA38-0682*. For complete and detailed information on using interlanguage calls, refer to *z/OS Language Environment Writing Interlanguage Communication Applications, SA38-0684*.

The following table lists the z/OS XL C/C++ and related documents. The table groups the documents according to the tasks they describe.

Table 3. Documents by task

Tasks	Documents
Planning, preparing, and migrating to z/OS XL C/C++	<ul style="list-style-type: none"> <li>• <i>z/OS XL C/C++ Compiler and Runtime Migration Guide for the Application Programmer, GC14-7306</i></li> <li>• <i>z/OS Language Environment Customization, SA38-0685</i></li> <li>• <i>z/OS Language Environment Runtime Application Migration Guide, GA32-0912</i></li> <li>• <i>z/OS UNIX System Services Planning, GA32-0884</i></li> <li>• <i>z/OS Planning for Installation, GA32-0890</i></li> </ul>
Installing	<ul style="list-style-type: none"> <li>• <i>z/OS Program Directory</i></li> <li>• <i>z/OS Planning for Installation, GA32-0890</i></li> <li>• <i>z/OS Language Environment Customization, SA38-0685</i></li> </ul>
Option customization	<ul style="list-style-type: none"> <li>• <i>z/OS XL C/C++ User's Guide, SC14-7307</i></li> </ul>
Coding programs	<ul style="list-style-type: none"> <li>• <i>z/OS XL C/C++ Runtime Library Reference, SC14-7314</i></li> <li>• <i>z/OS XL C/C++ Language Reference, SC14-7308</i></li> <li>• <i>z/OS XL C/C++ Programming Guide, SC14-7315</i></li> <li>• <i>z/OS Metal C Programming Guide and Reference, SC14-7313</i></li> <li>• <i>z/OS Language Environment Concepts Guide, SA38-0687</i></li> <li>• <i>z/OS Language Environment Programming Guide, SA38-0682</i></li> <li>• <i>z/OS Language Environment Programming Reference, SA38-0683</i></li> </ul>
Coding and binding programs with interlanguage calls	<ul style="list-style-type: none"> <li>• <i>z/OS XL C/C++ Programming Guide, SC14-7315</i></li> <li>• <i>z/OS XL C/C++ Language Reference, SC14-7308</i></li> <li>• <i>z/OS Language Environment Programming Guide, SA38-0682</i></li> <li>• <i>z/OS Language Environment Writing Interlanguage Communication Applications, SA38-0684</i></li> <li>• <i>z/OS MVS Program Management: User's Guide and Reference, SA23-1393</i></li> <li>• <i>z/OS MVS Program Management: Advanced Facilities, SA23-1392</i></li> </ul>
Compiling, binding, and running programs	<ul style="list-style-type: none"> <li>• <i>z/OS XL C/C++ User's Guide, SC14-7307</i></li> <li>• <i>z/OS Language Environment Programming Guide, SA38-0682</i></li> <li>• <i>z/OS Language Environment Debugging Guide, GA32-0908</i></li> <li>• <i>z/OS MVS Program Management: User's Guide and Reference, SA23-1393</i></li> <li>• <i>z/OS MVS Program Management: Advanced Facilities, SA23-1392</i></li> </ul>

Table 3. Documents by task (continued)

Tasks	Documents
Compiling and binding applications in the z/OS UNIX (z/OS UNIX) environment	<ul style="list-style-type: none"> <li>• <i>z/OS XL C/C++ User's Guide, SC14-7307</i></li> <li>• <i>z/OS UNIX System Services User's Guide, SA23-2279</i></li> <li>• <i>z/OS UNIX System Services Command Reference, SA23-2280</i></li> <li>• <i>z/OS MVS Program Management: User's Guide and Reference, SA23-1393</i></li> <li>• <i>z/OS MVS Program Management: Advanced Facilities, SA23-1392</i></li> </ul>
Debugging programs	<ul style="list-style-type: none"> <li>• README file</li> <li>• <i>z/OS XL C/C++ User's Guide, SC14-7307</i></li> <li>• <i>z/OS XL C/C++ Messages, GC14-7305</i></li> <li>• <i>z/OS XL C/C++ Programming Guide, SC14-7315</i></li> <li>• <i>z/OS Language Environment Programming Guide, SA38-0682</i></li> <li>• <i>z/OS Language Environment Debugging Guide, GA32-0908</i></li> <li>• <i>z/OS Language Environment Runtime Messages, SA38-0686</i></li> <li>• <i>z/OS UNIX System Services Messages and Codes, SA23-2284</i></li> <li>• <i>z/OS UNIX System Services User's Guide, SA23-2279</i></li> <li>• <i>z/OS UNIX System Services Command Reference, SA23-2280</i></li> <li>• <i>z/OS UNIX System Services Programming Tools, SA23-2282</i></li> <li>• Debug Tool documentation, available on the Debug Tool Library page on the World Wide Web (<a href="http://www.ibm.com/software/awdtools/debugtool/library/">http://www.ibm.com/software/awdtools/debugtool/library/</a>)</li> </ul>
Developing debuggers and profilers	<ul style="list-style-type: none"> <li>• <i>z/OS Common Debug Architecture User's Guide, SC14-7310</i></li> <li>• <i>z/OS Common Debug Architecture Library Reference, SC14-7311</i></li> <li>• <i>DWARF/ELF Extensions Library Reference, SC14-7312</i></li> </ul>
Packaging XL C/C++ applications	<ul style="list-style-type: none"> <li>• <i>z/OS XL C/C++ Programming Guide, SC14-7315</i></li> <li>• <i>z/OS XL C/C++ User's Guide, SC14-7307</i></li> </ul>
Using shells and utilities in the z/OS UNIX environment	<ul style="list-style-type: none"> <li>• <i>z/OS XL C/C++ User's Guide, SC14-7307</i></li> <li>• <i>z/OS UNIX System Services Command Reference, SA23-2280</i></li> <li>• <i>z/OS UNIX System Services Messages and Codes, SA23-2284</i></li> </ul>
Using sockets library functions in the z/OS UNIX environment	<ul style="list-style-type: none"> <li>• <i>z/OS XL C/C++ Runtime Library Reference, SC14-7314</i></li> </ul>
Using the ISO Standard C++ Library to write portable C/C++ code that complies with ISO standards	<ul style="list-style-type: none"> <li>• <i>Standard C++ Library Reference, SC14-7309</i></li> </ul>
Performing diagnosis and submitting an Authorized Program Analysis Report (APAR)	<ul style="list-style-type: none"> <li>• <i>z/OS XL C/C++ User's Guide, SC14-7307</i></li> </ul>

**Note:** For information on using the prelinker, see the appendix on prelinking and linking z/OS XL C/C++ programs in *z/OS XL C/C++ User's Guide*.

## Softcopy documents

The z/OS XL C/C++ publications are supplied in PDF format and available for download at <http://www.ibm.com/software/awdtools/czos/library/>.

To read a PDF file, use the Adobe Reader. If you do not have the Adobe Reader, you can download it (subject to Adobe license terms) from the Adobe website at <http://www.adobe.com>.

You can also browse the documents on the World Wide Web by visiting the z/OS library at <http://www.ibm.com/systems/z/os/zos/bkserv/>.

**Note:** For further information on viewing and printing softcopy documents and using IBM BookManager®, see *z/OS Information Roadmap*.

---

## **z/OS XL C/C++ on the World Wide Web**

Additional information on z/OS XL C/C++ is available on the World Wide Web on the z/OS XL C/C++ home page at <http://www.ibm.com/software/awdtools/czos/>.

This page contains late-breaking information about the z/OS XL C/C++ product, including the compiler, the C/C++ libraries, and utilities. There are links to other useful information, such as the z/OS XL C/C++ information library and the libraries of other z/OS elements that are available on the web. The z/OS XL C/C++ home page also contains links to other related websites.

## **Where to find more information**

For an overview of the information associated with z/OS, see *z/OS Information Roadmap*.

### **Information updates on the web**

For the latest information updates that have been provided in PTF cover letters and documentation APARs for z/OS, see the online document z/OS APAR book ([http://publibz.boulder.ibm.com/cgi-bin/bookmgr\\_OS390/Shelves/ZDOCAPAR](http://publibz.boulder.ibm.com/cgi-bin/bookmgr_OS390/Shelves/ZDOCAPAR)).

This document is updated weekly and lists documentation changes before they are incorporated into z/OS publications.

### **The z/OS Basic Skills Information Center**

The z/OS Basic Skills Information Center is a Web-based information resource intended to help users learn the basic concepts of z/OS, the operating system that runs most of the IBM mainframe computers in use today. The Information Center is designed to introduce a new generation of Information Technology professionals to basic concepts and help them prepare for a career as a z/OS professional, such as a z/OS system programmer.

Specifically, the z/OS Basic Skills Information Center is intended to achieve the following objectives:

- Provide basic education and information about z/OS without charge
- Shorten the time it takes for people to become productive on the mainframe
- Make it easier for new people to learn z/OS.

To access the z/OS Basic Skills Information Center, open your Web browser to the following Web site, which is available to all users (no login required): z/OS Basic Skills Information Center home page (<http://publib.boulder.ibm.com/infocenter/zos/basics/index.jsp>)

---

## **Technical support**

Additional technical support is available from the z/OS XL C/C++ Support page. This page provides a portal with search capabilities to a large selection of technical support FAQs and other support documents. You can find the z/OS XL C/C++ Support page on the Web at:



<http://www.ibm.com/software/awdtools/czos/support>

If you cannot find what you need, you can e-mail:

[compinfo@ca.ibm.com](mailto:compinfo@ca.ibm.com)

For the latest information about z/OS XL C/C++, visit the product information site at:

<http://www.ibm.com/software/awdtools/czos/>

For information about boosting performance, productivity and portability, visit the C/C++ Cafe at:

<http://www.ibm.com/software/rational/cafe/community/ccpp>

---

## How to send your comments to IBM

We appreciate your input on this publication. Feel free to comment on the clarity, accuracy, and completeness of the information or provide any other feedback that you have.

Use one of the following methods to send your comments:

1. Send an email to [mhvrcfs@us.ibm.com](mailto:mhvrcfs@us.ibm.com).
2. Send an email from the "Contact us" web page for z/OS (<http://www.ibm.com/systems/z/os/zos/webqs.html>).
3. Mail the comments to the following address:  
IBM Corporation  
Attention: MHVRCFS Reader Comments  
Department H6MA, Building 707  
2455 South Road  
Poughkeepsie, NY 12601-5400  
US
4. Fax the comments to us, as follows:  
From the United States and Canada: 1+845+432-9405  
From all other countries: Your international access code +1+845+432-9405

Include the following information:

- Your name and address.
- Your email address.
- Your telephone or fax number.
- The publication title and order number:  
z/OS V2R1.0 XL C/C++ Compiler and Runtime Migration Guide for the  
Application Programmer  
GC14-7306-01
- The topic and page number that is related to your comment.
- The text of your comment.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute the comments in any way appropriate without incurring any obligation to you.

IBM or any other organizations use the personal information that you supply to contact you only about the issues that you submit.

## If you have a technical problem

Do not use the feedback methods that are listed for sending comments. Instead, take one of the following actions:

- Contact your IBM service representative.
- Call IBM technical support.
- Visit the IBM Support Portal at z/OS support page (<http://www.ibm.com/systems/z/support/>).

---

## Part 1. Introduction

Before you start migrating applications to z/OS V2R1 XL C/C++, familiarize yourself with the following information:

- Chapter 1, “New migration issues for z/OS XL C/C+ V2R1M1,” on page 3
- Chapter 3, “Program migration checklists,” on page 9



---

## Chapter 1. New migration issues for z/OS XL C/C+ V2R1M1

The IBM z/OS XL C/C++ compiler updates for z/OS V2R1 are available as a web download from <http://www-03.ibm.com/systems/z/os/zos/downloads/> (see "XL C/C++ V2R1M1 web deliverable for z/OS 2.1"). These updates are referred to as z/OS XL C/C++ V2R1M1 in this document. Performance and usability enhancements are made for the z/OS V2R1 XL C/C++ compiler. For detailed information about these changes, see *z/OS XL C/C++ User's Guide, SC14-7307*.

This document alerts you to the migration issues that result from the following enhancements:

### New compiler options

Use the following new compiler options that are introduced with z/OS XL C/C++ V2R1M1 :

- ASM - Specify the ASM compiler option to support embedded assembler source inside the C and C++ programs.
- ASMLIB - Use the ASMLIB compiler option to specify assembler macro libraries to be used.
- FUNCEVENT - Specify the FUNCEVENT compiler option to enable LE CEL4CASR, CELHCASR, and \_\_CEL4CASR CWI (compler-writer interface) notifications for each specified function upon function entry.
- -MG, -MT, and -MQ - Specify the -MG, -MT, and -MQ compiler options to generate dependency files that can be used by the make utility.
- VECTOR - Specify the VECTOR compiler option to enable the vector processing support.

For detailed information about these new compiler options, see *z/OS XL C/C++ User's Guide, SC14-7307*.

### New compiler suboptions

Use the following compiler suboptions that are introduced with z/OS XL C/C++ V2R1M1:

- ARCH(11) - Specify this suboption to instruct the compiler to produce code that uses instructions available on the 2964-xxx (IBM z Systems™ z13™ (z13)) models in z/Architecture® mode.
- KEYWORD(asm) - Specify this suboption to add asm to the list of keywords for C and C++ languages.
- LANGLVL(REDEFMAC) - Specify this suboption to redefine a macro without a prior #undef or undefine() statement.
- MAKEDEP(GCC) - Specify this suboption to instruct the compiler to produce make dependencies file format with a single make rule for all dependencies.
- MAKEDEP(PPONLY) - Specify this suboption to instructs the compiler to produce only the make dependencies file without generating an object file, with the same make file format as the format produced with the gcc suboption.
- NAMEMANGLING(zOSV2R1M1\_ANSI) - Use this suboption to specify the name mangling scheme that is compatible with z/OS XL C++ V2R1M1 link modules.

- TUNE(11) - Specify this suboption to instruct the compiler to generate code that is optimized for the 2964-xxx (IBM z Systems z13 (z13)) models.

For detailed information about these suboptions, see *z/OS XL C/C++ User's Guide, SC14-7307*.

#### **Hardware model and feature built-ins**

Use the hardware model and feature built-ins in your source programs to query the model of the underlying hardware and the features unique to the identified model. For detailed information, see *Hardware model and feature built-ins in z/OS XL C/C++ Programming Guide, SC14-7315*.

#### **Support for inline assembly statement**

Inline assembly statement in source code is supported in C and C++ programs. The ASM option causes the `__asm` and `__asm__` statements to follow the asm statement rules. The ASMLIB option specifies the assembler macro libraries to be used when assembling the inlined statements in source code. For more information, see `ASM | NOASM` and `ASMLIB | NOASMLIB` in *z/OS XL C/C++ User's Guide, SC14-7307*.

When the ASM option is specified, the `__IBM_ASM_SUPPORT` macro is predefined to 1. The `__IBM_INLINE_ASM_SUPPORT` macro has been removed.

#### **Support for vector processing**

Explore the vector programming support to make use of the Vector Facility for z/Architecture. For detailed information, see *Using vector programming support in z/OS XL C/C++ Programming Guide, SC14-7315*.

#### **Support for MASS and ATLAS libraries**

Use the Mathematical Acceleration Subsystem (MASS) and Automatically Tuned Linear Algebra Software (ATLAS) libraries for high-performance mathematical computing. For detailed information, see *Using high performance libraries in z/OS XL C/C++ Programming Guide, SC14-7315*.

#### **Migration tools**

You can use migration tools to facilitate migration activities. For detailed information, see "Tools that facilitate your migration" on page 13.

---

## Chapter 2. New migration issues for z/OS V2R1 XL C/C++

IBM z/OS V2R1 XL C/C++ compiler has made performance and usability enhancements for the IBM z/OS operating platform V2R1 release ("z/OS V2R1" hereafter). For detailed information about these changes, see *z/OS XL C/C++ User's Guide, SC14-7307*.

For information about the changes that the IBM Language Environment element has made for z/OS V2R1, see "What's New in Language Environment for z/OS" in *z/OS Language Environment Concepts Guide*.

This document alerts you to the migration issues that result from the following enhancements:

### New compiler options

z/OS V2R1 XL C/C++ compiler introduces support for the following new compiler options:

- INCLUDE | NOINCLUDE - Specify the INCLUDE compiler option to include additional header files in your compilation unit.
- SMP | NOSMP - Specify the SMP compiler option to enable parallelization of your program code.
- SYSSTATE - Specify the SYSSTATE compiler option to include additional parameters to the SYSSTATE macro that is generated by the compiler.
- THREADED | NOTHREADED - Specify the THREADED compiler option to maintain thread safety of your code when compiling or linking multithreaded applications.

For detailed information about these new compiler options, see *z/OS XL C/C++ User's Guide, SC14-7307*.

### New compiler suboptions

z/OS V2R1 XL C/C++ compiler introduces support for the following new compiler suboptions:

- ARCH(10) - Specify this suboption to instruct the compiler to produce code that uses instructions available on the 2827-xxx models (IBM zEnterprise® EC12 (zEC12)) in z/Architecture mode.
- DEBUG(LEVEL(*level*)) - Specify debug levels in the range of 0 - 9 to control the amount of debug information produced.
- LANGLVL(C1XNORETURN | CONSTEXPR | DEFAULTANDDELETE | EXTC1X | EXPLICITCONVERSIONOPERATORS | REFERENCECOLLAPSING | RIGHTANGLEBRACKET | RVALUEREFERENCES | SCOPEENUM) - Specify these suboptions to invoke the C11 and C++11 features that are currently supported by the compiler and other implementation-specific language extensions.
- NAMEMANGLING(zOSV2R1\_ANSI) - Use this suboption to specify the name mangling scheme that is compatible with z/OS XL C++ V2R1 link modules.
- NOSUPPRESS(*message\_identifier*) - Specify message identifiers with the NOSUPPRESS option to disable previous SUPPRESS instances with the same message identifiers.

- OBJECTMODEL(CLASSIC) - Specify OBJECTMODEL(CLASSIC) instead of OBJECTMODEL(COMPAT).
- STRICT(SUBSCRIPTWRAP) - Specify this suboption to prevent the compiler from assuming that array subscript expressions will never overflow.
- TARGET(zOSV2R1) - Specify this suboption to instruct the compiler to generate object code to run under z/OS Version 2 Release 1 and subsequent releases.
- TUNE(10) - Specify this suboption to instruct the compiler to generate code that is optimized for the 2827-xxx (IBM zEnterprise EC12 (zEC12)) models.

For detailed information about these suboptions, see *z/OS XL C/C++ User's Guide, SC14-7307*.

### **New hardware built-in functions**

z/OS V2R1 XL C/C++ compiler introduces support for the following hardware packed-decimal instructions:

- Compare Decimal (CP)
- Add Decimal (AP)
- Subtract Decimal (SP)
- Multiply Decimal (MP)
- Divide Decimal (DP)
- Shift and Round Decimal (SRP)

These hardware built-in functions are available on all ARCH levels, and the function prototypes are declared in the `builtins.h` header file.

z/OS V2R1 XL C/C++ compiler introduces support for the following hardware instructions for conversions between DFP and Zoned types:

- Zoned to DFP conversion (CDZT)
- Zoned to DFP conversion (CXZT)
- DFP to Zoned conversion (CZDT)
- DFP to Zoned conversion (CZXT)

These hardware built-in functions can be used only when both the DFP and ARCH(10) compiler options are in effect. For detailed information about these hardware built-in functions, see *Using hardware built-in functions in z/OS XL C/C++ Programming Guide*.

### **A type that is denoted in an exception-specification should not denote an incomplete type**

Starting with z/OS V2R1, this is diagnosed for exception-specifications in additional contexts, including function declarations. For detailed information, see "Issues caused by the use of incomplete types in exception-specifications" on page 128.

### **Behavior change of the CONVLIT compiler option**

Starting with z/OS V2R1, the CONVLIT option with empty suboption list, CONVLIT() or `-qconvlit=`, preserves any previous settings of the suboptions, and the compiler does not use the default codepage, or the codepage specified by the LOCALE option as in prior releases. Do not specify the CONVLIT option without suboptions unless needed.



### **Behavior change of the IPA compiler option**

Starting with z/OS V2R1, when the compiler is invoked from JCL, the default optimization level for the IPA option is OPTIMIZE(2).

### **Behavior change of the MARGINS and SEQUENCE compiler options**

Starting with z/OS V2R1, when the compiler is invoked in USS, the default behavior of the MARGINS and SEQUENCE options is changed to match the behavior in JCL invocation. The compiler no longer treats default values of these options as if they were explicitly specified on the command line. To have the old behavior of these options, specify NOMARGINS and NOSEQUENCE on the command line or in the xlc configuration file.

Starting with z/OS V2R1, when the MARGINS option is specified without suboptions for C++ compilation, the default values for the first and last column of the source input are 1 and 32760 respectively. Prior to z/OS V2R1, the values were 1 and 72.

### **Changes to default ARCH and TUNE level**

Starting with z/OS V2R1, the default ARCH level is changed from ARCH(5) to ARCH(7), and the default TUNE level is changed from TUNE(5) to TUNE(7). For Metal C compilation commands, if a lower ARCH level is required, you must explicitly specify the lowest ARCH level that the program could run at. For detailed information, see ARCHITECTURE and TUNE in *z/OS XL C/C++ User's Guide, SC14-7307*.

### **Complex type initialization**

z/OS V2R1 XL C/C++ compiler supports the initialization of complex types.

For C, macros *CMPLX*, *CMPLXF*, and *CMPLXL* are defined inside the standard header file *complex.h* to enable the initialization of complex types under the EXTC1X language level. For more information, see Initialization of complex types (C11) in *z/OS XL C/C++ Language Reference*.

For C++, you can specify the option *LANGLVL(COMPLEXINIT)* to initialize complex types. For more information, see *LANGLVL* compiler option in *z/OS XL C/C++ User's Guide, SC14-7307*.

### **Name change of suboption OBJECTMODEL(COMPAT)**

Starting with z/OS V2R1, the name of suboption *OBJECTMODEL(COMPAT)* is changed to *OBJECTMODEL(CLASSIC)*. Use *CLASSIC* instead of *COMPAT*, although *COMPAT* is still accepted as the synonym of *CLASSIC*.

### **New debug information format**

Starting with z/OS V2R1, when the *DEBUG* option is in effect, the compiler generates debug information based on the DWARF Version 4 debugging information format.

### **Support for parallel programming**

The z/OS V2R1 XL C/C++ compiler introduces support for OpenMP parallelization directives. You can specify the *SMP* compiler option to enable parallelization of your program code.

### **Migration tools**

You can use migration tools to facilitate migration activities. For detailed information, see "Tools that facilitate your migration" on page 13.



---

## Chapter 3. Program migration checklists

This information includes checklists that you can use at various stages of migrating an application to the z/OS V2R1 XL C/C++ compiler. These phases are:

- “Before you start your migration”
- “When you are compiling code” on page 10
- “When you are binding program objects or load modules” on page 11
- “When you are running an application” on page 12

For product history information to help you determine which topics in this document apply to your migration, see “Applicability of product information” on page 14.

---

### Before you start your migration

Before you migrate programs or applications to z/OS V2R1 XL C/C++ compiler, determine potential problems with your source code by reviewing the following checklist:

1. Determine the group of compiler releases from which you are migrating:
  - An earlier z/OS C/C++ compiler
  - An OS/390 C/C++ compiler
  - A pre-OS/390 C/C++ compiler
2. View the documentation updates and other post-release information provided by the ReadMe files at <http://www.ibm.com/support/docview.wss?uid=swg27007531>.
3. Review the changes introduced in z/OS V2R1 XL C/C++ compiler. See Chapter 1, “New migration issues for z/OS XL C/C+ V2R1M1,” on page 3.
4. Review the changes that have been implemented since the last C/C++ compiler that was used with the application:
  - If you are migrating from an earlier z/OS C/C++ application, see Part 4, “Migration of earlier z/OS C/C++ applications to z/OS V2R1 XL C/C++,” on page 77.
  - If you are migrating from an OS/390 C/C++ application, see Part 3, “Migration of OS/390 C/C++ applications to z/OS V2R1 XL C/C++,” on page 57.
  - If you are migrating from a pre-OS/390 C/C++ compiler, see Part 2, “Migration of pre-OS/390 C/C++ applications to z/OS V2R1 XL C/C++,” on page 17.
5. Review the types of source code changes that have been identified since the last C/C++ compiler that was used with the application:
  - If you are migrating from an earlier z/OS C/C++ application, see Chapter 14, “Source code compatibility issues with earlier z/OS C/C++ programs,” on page 79.
  - If you are migrating from an OS/390 C/C++ application, see Chapter 9, “Source code compatibility issues with OS/390 programs,” on page 59.
  - If you are migrating from a pre-OS/390 C/C++ application, see Chapter 4, “Source code compatibility issues with pre-OS/390 C/C++ programs,” on page 19.

**Note:** If your application uses class libraries that have been modified or are no longer supported, the resulting migration issues are discussed as source code compatibility changes.

6. Use the INFO compiler option to identify the following potential problems:
  - Functions not prototyped. See “INFO compiler option” on page 64.

**Notes:**

- a. Function prototypes allow the compiler to check for mismatched parameters.
  - b. Return parameters might be mis-matched, especially when the code expects a pointer. (For example, malloc and family)
- Assignment of a long or a pointer to an integer, or assignment of an integer to a pointer. See “Pointer incompatibilities” on page 21.

**Note:** This type of assignment could cause truncation. A reference to the pointer might be invalid. Even assignments with an explicit cast will be flagged. See “CHECKOUT(CAST) compiler option” on page 64.

7. If your code must be compliant with a specific ISO C++ standard, see Part 5, “ISO Standard C++ compliance migration issues,” on page 115.
8. If you are using the IBM object model for an XL C++ program or application that was last compiled or executed with the compat object model, see “Alignment incompatibilities between object models” on page 102.

---

## When you are compiling code

Before you use z/OS V2R1 XL C/C++ compiler to compile pre-existing source code, review the following checklist:

1. Review the compile-time migration issues that have been identified in one of the following topics:
  - Chapter 15, “Compile-time migration issues with earlier z/OS C/C++ programs,” on page 87.
  - Chapter 10, “Compile-time migration issues with OS/390 programs,” on page 61.
  - Chapter 5, “Compile-time issues with pre-OS/390 C/C++ programs,” on page 25.
2. If you are using a SYSLIB DD card to compile your XL C/C++ program, see “Changes that affect SYSLIB DD cards” on page 30.
3. If your XL C/C++ program behaves unexpectedly after you re-compile it, consider the following possibilities:
  - At least one of the compiler options that you used does not function as it did before, or it is no longer supported. See the appropriate information in this document:
    - If you are migrating from any application, see “Changes in compiler option functionality” on page 91
    - If you are migrating from an OS/390 C/C++ application, see “Changes in compiler options” on page 62
    - If you are migrating from a pre-OS/390 C/C++ application, see “Changes in compiler options” on page 25
  - The compiler invocation has been modified since you last used it.
  - There might be a newer option or invocation that is more suitable for your source program. See the appropriate information in this document:

- If you are migrating from any application, see “Changes that affect compiler invocations” on page 95
  - If you are migrating from a pre-OS/390 C/C++ application, see “Changes that affect compiler invocations” on page 29
4. Are you using the NAMEMANGLING compiler option under ILP32 in a batch environment? If so, see “ILP32 compiler option and name mangling” on page 98.
  5. If you are using the IPA or IPA(LINK) option to compile the program, see the appropriate information in this document:
    - If you are migrating from any application, see:
      - “Changes that affect JCL procedures” on page 97
      - “IPA(LINK) compiler option and exploitation of 64-bit virtual memory” on page 99
    - If you are migrating from a pre-OS/390 C/C++ application, see
      - “IPA Link step default changes” on page 65
      - “IPA object module binary compatibility” on page 66

---

## When you are binding program objects or load modules

Before you try to bind or relink pre-existing program objects or load modules, review the following checklist:

1. Review the potential bind-time migration issues that have been identified since the last C/C++ compiler that was used with the application:
  - If you are migrating from any z/OS C/C++ application, see Chapter 16, “Bind-time migration issues with earlier z/OS C/C++ programs,” on page 101.
  - If you are migrating from an OS/390 C/C++ application, see Chapter 11, “Bind-time migration issues with OS/390 C/C++ programs,” on page 71.
  - If you are migrating from a pre-OS/390 C/C++ application, see Chapter 6, “Bind-time migration issues with pre-OS/390 C/C++ programs,” on page 31.
2. Consider the following questions:
  - Are there any relevant library changes? For information, see Chapter 13, “Migration issues resulting from class library changes between OS/390 C/C++ applications and Standard C++ library,” on page 75.
  - Do input/output or other operations have library dependencies that might be affected by product changes since the program was last run? For more information, see Chapter 8, “Input and output operations compatibility,” on page 51.
  - Has there been any change in exception handling since the program was last run? For information, see “Hardware and OS exceptions” on page 48 or (for C++ programs) “cv-qualifications when the thrown and caught types are the same” on page 122.
  - Are you using System Program C (SPC) facility modules? For information, see “Assembler source code changes in System Programming C (SPC) applications built with EDCXSTRX” on page 22.
  - Does the program need to access IBM CICS or IBM DB2 data? For information, see Part 6, “Migration issues for C/C++ applications that use other IBM products,” on page 131.
  - Does the C or C++ module include interlanguage calls (ILC)? For information, see “C/370 modules with interlanguage calls (ILC)” on page 34 or more specific topics listed in the index.

If you are migrating from a pre-OS/390 C/C++ application, are you using the TARGET(OSV2R10) compiler option? If so, see “Namespace pollution binder errors” on page 101.

---

## When you are running an application

Before you try to run a legacy application under z/OS V2R1, review the following checklist:

1. Review the potential runtime migration issues that have been identified:
  - If the application has been run successfully under an earlier z/OS runtime environment, see Chapter 17, “Runtime migration issues with earlier z/OS C/C++ applications,” on page 105.
  - If the application was last run successfully under an OS/390 runtime environment, see Chapter 12, “Runtime migration issues with OS/390 C/C++ applications,” on page 73.
  - If the application has not been run in an environment more recent than an OS/390 runtime environment, see Chapter 7, “Runtime migration issues with pre-OS/390 C/C++ applications,” on page 41.
2. If you need to retain the runtime behavior of the application, see “Retention of previous runtime behavior” on page 106, “Retention of OS/390 runtime behavior” on page 73, or “Retention of pre-OS/390 runtime behavior” on page 41, as appropriate.
3. If you are migrating from a runtime environment that predates the z/OS V1R5 Language Environment release, verify the following:
  - The concatenation order of your libraries, to ensure that there are no links to non-Language Environment interfaces.
  - Data set names that are referenced by all customized procedures (such as JCL and makefiles) have not been changed.

See “Runtime library compatibility issues with pre-OS/390 applications” on page 45 and “Changes that affect customized JCL procedures” on page 43.

4. If your application does not run, it may be either a migration problem, or an error in your program that surfaces as a result of enhancements to Language Environment services. Do the following:

- Relink application load modules or program objects if any of the following are true:

It is an IBM C/370™ application.

It contains ILCs between C and Fortran, or between C and COBOL. For information, see “C/370 modules with interlanguage calls (ILC)” on page 34.

It is an SPC application that uses the library. For information, see “Assembler source code changes in System Programming C (SPC) applications built with EDCXSTRX” on page 22.

It contains calls to `ctest()`. For information, see “Requirements for relinking C/370 modules that invoke Debug Tool” on page 34.

The PDS with the low-level qualifier SCEERUN (which belongs to the runtime library), is not concatenated ahead of the PDS with the low-level qualifier SIBMLINK (which belongs to the C-PL/I Common Library). For information, see “Common library initialization compatibility issues with C/370 modules” on page 46.

A message suggests either resetting an environment variable or relinking application load modules or program objects. For information, see

Chapter 16, “Bind-time migration issues with earlier z/OS C/C++ programs,” on page 101, “Runtime library messages” on page 41 or “Program modules from an earlier release” on page 101.

- Use the STORAGE and HEAP runtime options to find uninitialized storage. For information about initialization schemes and procedures, see “Common library initialization compatibility issues with C/370 modules” on page 46.

**Notes:**

- a. In some cases, applications will run with uninitialized storage, because the runtime library may inadvertently clear storage, or because the storage location referenced is set to zero.
  - b. IBM recommends STORAGE(FE,DE,BE) and HEAP(16,16,ANY,FREE) to determine if your application is coded correctly. Any uninitialized pointers will fail at first reference instead of accidentally referencing storage locations at random.
  - c. The STORAGE or HEAP option will cause your program to run more slowly. Do not use them for production; use them for development only.
- Look for undocumented interfaces.  
It is possible that your application has dependencies on undocumented interfaces. For example, you might have dependencies on library control blocks, specific errno values, or specific return values. Alter your code to use only documented interfaces, and then recompile the code and relink the load modules or program objects. For information, see Chapter 8, “Input and output operations compatibility,” on page 51.
  - It is possible that your application is being initialized or terminated differently because of changes in the runtime environment. See “Common library initialization compatibility issues with C/370 modules” on page 46 and “Order of destruction for statically initialized objects” on page 120.
5. If your application does not require the features provided by z/OS V2R1, use environment variables to maintain the expected behavior. For information, see “Changes that affect compiler invocations” on page 95.
  6. Contact your System Programmer to determine whether or not all service has been applied to your system. Often, the problem you encounter has already been reported to IBM, and a fix is available.
  7. If you have verified with your System Programmer that all service has been applied to your system, ask your Service Representative to open a Problem Management Record (PMR) against the applicable IBM product. For information on how to open a PMR, refer to <http://techsupport.services.ibm.com/guides/handbook.html>.

---

## Tools that facilitate your migration

This section describes tools available for your assistance during the migration activity.

### The Edge Portfolio Analyzer

The Edge Portfolio Analyzer can provide assistance in taking an inventory of your existing XL C/C++ load modules. The object must be compiled with z/OS V1R10 XL C/C++ compiler or later for reporting of compiler options.

The Edge Portfolio Analyzer is no longer sold by IBM. For more information about the Edge Portfolio Analyzer, visit their Web site at <http://www.edge-information.com>.

**Note:** Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

Neither International Business Machines Corporation nor any of its affiliates assume any responsibility or liability in respect of any results obtained by implementing any recommendations contained in this article/document. Implementation of any such recommendations is entirely at the implementor's risk.

## Applicability of product information

In Table 4, references to the products listed in the first column also apply to the products in the second column.

*Table 4. Product references*

Referenced compilers	Related products
Pre-OS/390 C/C++ compilers  <b>Note:</b> If you are migrating a program that has been run successfully only with a pre-OS/390 C/C++ compiler, contact your service representative.	<ul style="list-style-type: none"> <li>• IBM C/C++ for MVS/ESA V3R1 or V3R2</li> <li>• IBM AD/Cycle C/370 V1R1 or V1R2</li> <li>• IBM C/370 V1R1 or V1R2</li> <li>• IBM C/370 V2R1 compiler and the IBM C/370 V2R1 library</li> <li>• IBM C/370 V2R1 compiler and the IBM C/370 V2R2 library</li> </ul>
OS/390 C/C++ compilers  <b>Notes:</b> <ol style="list-style-type: none"> <li>1. IBM OS/390 V1R1 C/C++ is the same as IBM C/C++ for MVS/ESA V3R2.</li> <li>2. IBM z/OS V1R1 C/C++ is the same as IBM OS/390 V2R10 C/C++. IBM OS/390 V2R10 is also reshipped in z/OS V1R2 through to V1R6.</li> <li>3. If you are migrating a program that has been run successfully only with the OS/390 V1R1 C/C++ compiler, contact your service representative.</li> <li>4. IBM OS/390 is no long in service.</li> </ol>	<ul style="list-style-type: none"> <li>• IBM OS/390 V1R1 C/C++ (reship of IBM C/C++ for MVS/ESA V3R2)</li> <li>• IBM OS/390 V1R2 or V1R3 C/C++</li> <li>• IBM OS/390 V2R4, V2R5, V2R6, V2R7, V2R8, V2R9, or V2R10 C/C++</li> <li>• IBM z/OS V1R1 C/C++ (reship of IBM OS/390 V2R10 C/C++)</li> </ul>
Earlier releases of the z/OS C/C++ compilers <b>Note:</b> Service is available for compilers z/OS XL C/C++ V1R12 through z/OS V2R1 XL C/C++.	<ul style="list-style-type: none"> <li>• IBM z/OS V1R1 C/C++ (equivalent to the OS/390 V2R10 compiler)</li> <li>• IBM z/OS V1R2 C/C++</li> <li>• IBM z/OS V1R3 C/C++</li> <li>• IBM z/OS V1R4 C/C++</li> <li>• IBM z/OS V1R5 C/C++</li> <li>• IBM z/OS V1R6 C/C++</li> <li>• IBM z/OS V1R7 XL C/C++</li> <li>• IBM z/OS V1R8 XL C/C++</li> <li>• IBM z/OS V1R9 XL C/C++</li> <li>• IBM z/OS V1R10 XL C/C++</li> <li>• IBM z/OS V1R11 XL C/C++</li> <li>• IBM z/OS V1R12 XL C/C++</li> <li>• IBM z/OS V1R13 XL C/C++</li> </ul>



## Version history of IBM C/C++ compilers and libraries

You can use the version history of IBM C/C++ compilers and libraries to help determine whether specific information in this document applies to your migration, as well as whether the information you seek is provided by this document.

The version history pertains to each C/370, VM/ESA, VSE/ESA, MVS/ESA, OS/390, z/OS and z/VM<sup>®</sup> compiler that has been distributed by IBM. It contains the following information:

- Compiler name and ID
- General release, end-of-marketing, and end-of-service dates
- Runtime library

**Note:** For the version history of IBM C/C++ compilers and libraries, see [http://www-03.ibm.com/systems/z/os/zos/support/zos\\_eos\\_dates.html](http://www-03.ibm.com/systems/z/os/zos/support/zos_eos_dates.html).



---

## Part 2. Migration of pre-OS/390 C/C++ applications to z/OS V2R1 XL C/C++

Prior to IBM OS/390, C/C++ applications were created with one of the following products:

- IBM C/C++ for MVS/ESA V3R1 or V3R2
- IBM AD/Cycle C/370 V1R1 or V1R2
- IBM C/370 V1R1 or V1R2
- IBM C/370 V2R1 compiler and the IBM C/370 V2R1 library
- IBM C/370 V2R1 compiler and the IBM C/370 V2R2 library

### Notes:

1.

If your application uses IBM CICS information or statements, also see Chapter 20, “Migration issues with earlier C/C++ applications that run CICS statements,” on page 133.

2.

If your application uses IBM DB2 information or statements, also see Chapter 21, “Migration issues with earlier C/C++ applications that use DB2 Universal Database,” on page 139.

The following topics provide information relevant to migrating a pre-OS/390 application to z/OS V2R1 XL C/C++:

- Chapter 4, “Source code compatibility issues with pre-OS/390 C/C++ programs,” on page 19
- Chapter 5, “Compile-time issues with pre-OS/390 C/C++ programs,” on page 25
- Chapter 6, “Bind-time migration issues with pre-OS/390 C/C++ programs,” on page 31
- Chapter 7, “Runtime migration issues with pre-OS/390 C/C++ applications,” on page 41
- Chapter 8, “Input and output operations compatibility,” on page 51



---

## Chapter 4. Source code compatibility issues with pre-OS/390 C/C++ programs

When you migrate applications that predate IBM OS/390 C/C++ compilers to the IBM z/OS V2R1 XL C/C++ product, be aware of the following migration issues:

- “Removal of IBM Open Class Library support”
- “Source code modifications necessitated by changes in runtime library”
- “Resource allocation and memory management issues”
- “Addressing incompatibilities” on page 20
- “Data type incompatibilities” on page 21
- “Changes required by programs with interlanguage calls” on page 22
- “Internationalization incompatibilities” on page 23

**Note:** Some source code compatibility issues can be addressed by modifying runtime options. See Chapter 12, “Runtime migration issues with OS/390 C/C++ applications,” on page 73.

---

### Removal of IBM Open Class Library support

As of z/OS V1R9, IBM Open Class<sup>®</sup> Library (IOC) dynamic link libraries (DLLs) are no longer shipped with the z/OS XL C/C++ compiler.

Any source dependency on an IOC DLL must be removed.

For information about the libraries that are supported by the current release, see *z/OS XL C/C++ Runtime Library Reference*.

---

### Source code modifications necessitated by changes in runtime library

When you migrate programs to z/OS V2R1 XL C/C++, review “Changes in runtime option specification” on page 43 for changes that will necessitate changes in your source code. Also review your use of the **#pragma runopts** directive in your source code.

#### The **#pragma runopts** directive

If occurrences of the ISASIZE/ISAINC, STAE/SPIE, LANGUAGE, or REPORT runtime options are specified by a **#pragma runopts** directive in your source code, you might want to change them to the supported equivalent before recompiling to avoid a warning or informational message during compilation.

For more information on preprocessor directives, refer to *z/OS XL C/C++ Language Reference*.

---

### Resource allocation and memory management issues

Incompatibilities in resource allocation and memory management might cause unexpected results in the output of your program. In your source code, you should be aware of potential problems when you use any of the following operators or structures:

- “The sizeof operator applied to a function return type” on page 20

- “A user-defined global new operator and array new”

## The sizeof operator applied to a function return type

Figure 1 illustrates how the behavior of `sizeof`, when applied to a function return type, was changed in the C/C++ for MVS/ESA V3R2 compiler.

```
char foo();
..
s = sizeof foo();
```

Figure 1. Statements that apply the `sizeof` operator to a function return type

If the example in Figure 1 is compiled with a compiler prior to C/C++ for MVS/ESA V3R2 compiler, `char` is widened to `int` in the return type, so `sizeof` returns `s = 4`.

If the example in Figure 1 is compiled with the C/C++ for MVS/ESA V3R2 compiler, or with any OS/390 C/C++ compiler, the size of the original `char` type is retained. In Figure 1, `sizeof` returns `s = 1`. The size of the original type of other data types such as `short`, and `float` is also retained.

If your code has a dependency on the behavior of the `sizeof` operator, be aware that with the OS/390 V2R4 C/C++ and subsequent compilers, you can use the `#pragma wsizeof` directive or the `WSIZEOF` compiler option to get `sizeof` to return the widened size for function return types.

For more information on `#pragma wsizeof`, see *z/OS XL C/C++ Language Reference, SC14-7308*. For more information on the `WSIZEOF` compiler option, see *z/OS XL C/C++ User's Guide, SC14-7307*.

## A user-defined global new operator and array new

If you are migrating from the C/C++ for MVS/ESA V3R2 compiler to z/OS V2R1 XL C/C++, and you have written your own global `new` operator, it is no longer called when you create an array object: In this case, you must add a local overloaded operator.

```
void* operator new (size_t sz) {
    g_new_count++;
    return MyMalloc(sz);
}

main() {
    X new_array[10]; // the global new operator
                    // shown above is not called
                    // in compilers for OS/390 or later
}
```

Figure 2. Example of user-defined global `new` operator and array `new`

---

## Addressing incompatibilities

Addressing incompatibilities might cause unexpected results in the output of your program. In your source code, you should be aware of the following migration issues:

- “C/370 V2 main program and main entry point” on page 21
- “Pointer incompatibilities” on page 21

## C/370 V2 main program and main entry point

C/370 V2 programs that are fetched must be recompiled without a main entry point. Any attempt to fetch a main program will fail.

## Pointer incompatibilities

According to the *ISO C Standard*, pointers to void types and pointers to functions are incompatible types. The C/370, AD/Cycle C/370, IBM C/MVS™, and z/OS XL C compilers perform some type-checking, such as in assignments, argument passing on function calls, and function return codes.

**Note:** If you are not conforming to ISO rules for the use of pointer types, your runtime results may not be as expected, especially when you are using the OPTIMIZE compiler option.

With the AD/Cycle C/370, and the C/C++ for MVS/ESA compilers, you could not assign NULL to an integer value. The statement shown in Figure 3 was not allowed:

```
int i = NULL;
```

Figure 3. Assignment of NULL to an integer value

With the z/OS XL C compilers, you can assign NULL pointers to void types only if you specify LANGLVL(COMMONC) when you compile your program. For information about constructs supported by LANGLVL(COMMONC) but not by LANGLVL(EXTENDED) or LANGLVL(ANSI), refer to LANGLVL compiler option in *z/OS XL C/C++ User's Guide, SC14-7307*.

---

## Data type incompatibilities

Data type incompatibilities might cause unexpected results in the output of your program. In your source code, you should be aware of potential migration issues:

- “Assignment restrictions for packed structures and unions”
- “DSECT header files and packed structures”

### Assignment restrictions for packed structures and unions

With the z/OS XL C compiler, you can no longer do the following:

- Assign packed and non-packed structures to each other.
- Assign packed and non-packed unions to each other.
- Pass a packed union or packed structure as a function parameter if a non-packed version is expected.
- Pass a non-packed union or non-packed structure as a function parameter if a packed version is expected.

If you attempt to do so, the compiler issues an error message.

### DSECT header files and packed structures

Header files generated by the DSECT utility use **#pragma pack** rather than the `_Packed` qualifier to pack structures or unions. In rare cases, you might have to modify and recompile your code.

**Note:** The `_Packed` qualifier is an IBM extension of the C language that was introduced with the C/370 family of compilers. It can also be applied to C++

classes. If you specify the `_Packed` qualifier on a structure or union that contains another structure or union as a member, the qualifier is not passed to the contained structure or union.

---

## Changes required by programs with interlanguage calls

If your code calls functions that have mixed-language input or output, you should be aware of the following potential source code issues:

- “Explicit program mask manipulations”
- “Assembler source code changes in System Programming C (SPC) applications built with EDCXSTRX”

### Explicit program mask manipulations

Programs created with the C/370 V2 compiler and library that explicitly manipulated the program mask might require source changes.

Changes are required if you have one of the following types of programs:

- A C program containing interlanguage calls (ILC), where the invoked code uses the S/370 decimal instructions that might generate an unmasked decimal overflow condition, requires modification for migration. Use either of the following two methods:
  - Preferred method: If the called routine is assembler, save the existing mask, set the new value, and when finished, restore the saved mask.
  - Change the C code so that the produced SIGFPE signal is ignored in the called code. In the following example, the SIGNAL calls surround the overflow-producing code. The SIGFPE exception handling is disabled before the problem signal is encountered, and then reenabled after it has been processed. See Figure 4.
- A C program containing assembler ILCs that explicitly alter the program mask, and do not explicitly save and restore it, also requires modification for migration.

If user code explicitly alters the state of the program mask, the value before modification must be saved, and the value restored to its former value after the modification. You must ensure that the decimal overflow program mask bit is enabled during the execution of C code. Failure to preserve the mask may result in unpredictable behavior.

These changes also apply in a System Programming C environment, and to Customer Information Control System (CICS) programs in the handling and management of the PSW mask.

```
signal(SIGFPE, SIG_IGN); /* ignore exceptions */
...
callit():                /* in called routine */
...
signal(SIGFPE, SIG_DFL); /* restore default handling */
```

*Figure 4. Statements that ignore SIGFPE exception and restore default exception handling*

### Assembler source code changes in System Programming C (SPC) applications built with EDCXSTRX

If you have SPC applications that are built with EDCXSTRX and use dynamic C library functions, note that the name of the C library function module was changed from EDCXV in C/370 V2 to CEEEV003 with the Language Environment V1R5



release. Change the name from EDCXV to CEEEV003 in the assembler source of your program that loads the library, and reassemble.

---

## Internationalization incompatibilities

If your code will be used with different locales, you should be aware of the information in “Support of alternate code points.”

### Support of alternate code points

The following alternate code points are not supported by z/OS V2R1 XL C/C++:

- X'8B' as alternate code point for X'C0' (the left brace)
- X'9B' as alternate code point for X'D0' (the right brace)

These alternate code points are supported by the C/370 and AD/Cycle C/370 compilers (the NOLOCALE option is required if you are using the AD/Cycle C/370 V1R2 compiler).

For more information about using coded character sets and locale functions, see *z/OS XL C/C++ Programming Guide, SC14-7315*.



---

## Chapter 5. Compile-time issues with pre-OS/390 C/C++ programs

When you use z/OS V2R1 XL C/C++ to compile programs that were last compiled as part of a pre-OS/390 C/C++ application, be aware of the following migration issues:

- “Changes in compiler listings, messages, and return codes”
- “Changes in compiler options”
- “Changes that affect compiler invocations” on page 29
- “Changes that affect SYSLIB DD cards” on page 30

---

### Changes in compiler listings, messages, and return codes

From release to release, message contents can change and, for some messages, return codes can change. Errors can become warnings, and warnings can become errors. You must update any application that is affected by changes in message contents or return codes. Do not build dependencies on message contents, message numbers, or return codes. See *z/OS XL C/C++ Messages* for a list of compiler messages.

Listing formats, especially the pseudo-assembler parts, will continue to change from release to release. Do not build dependencies on the structure or content of listings. For information about C listings or the C++ listings for the current release, refer to *z/OS XL C/C++ User's Guide, SC14-7307*.

### Macro redefinitions might result in severe errors

As of z/OS V1R7 XL C, the behavior of macro redefinition has changed. For certain language levels, the XL C compiler will issue a severe error message instead of a warning message when a macro is redefined to a value that is different from the first definition.

For information about the language levels that are affected, see “LANGLVL(ANSI), LANGLVL(SAA), or LANGLVL(SAAL2) compiler option and macro redefinitions” on page 27 and “LANGLVL(EXTENDED) compiler option and macro redefinitions” on page 27.

---

### Changes in compiler options

This topic describes changes that would affect your use of compiler options.

#### Compiler options that are no longer supported

This topic lists compiler options that were supported in pre-OS/390 compilers but not in subsequent compilers.

##### DECK compiler option

As of z/OS V1R2 C/C++ compiler, the DECK compiler option is no longer supported. If you want to route output to DD:SYSPUNCH, use OBJECT(DD:SYSPUNCH).

### **LANGLVL(COMPAT) compiler option**

In C/C++ for MVS/ESA V3R2, the LANTLRVL(COMPAT) option directed the compiler to generate code that is compatible with older levels of C and C++. As of z/OS V1R2 C/C++ compiler, the LANTLRVL(COMPAT) compiler option is no longer supported.

### **OMVS compiler option**

As of z/OS V1R2 C/C++ compiler, the OMVS compiler option is no longer supported. The replacement for it is the OE option.

### **SRCMSG compiler option**

As of z/OS V1R2 C/C++ compiler, the SRCMSG compiler option is no longer supported.

### **SYSLIB, USERLIB, SYSPATH and USERPATH compiler options**

In IBM C/C++ for MVS/ESA V3R2 compiler, the SYSLIB, USERLIB, SYSPATH and USERPATH compiler options directed the compiler to specified include files. As of z/OS V1R2 C/C++ compiler, these compiler options are no longer supported. Instead, use the SEARCH and LSEARCH options to find include files.

## **Compiler options that were introduced in OS/390 C/C++ or later**

When you are compiling pre-OS/390 C/C++ source code, you should treat compiler options that were introduced in OS/390 or later as new compiler options.

### **ENUMSIZE compiler option**

As of z/OS V1R7 XL C/C++, selected enumerated (enum) type declarations in system header files are protected to avoid potential execution errors. This allows you to specify the ENUMSIZE compiler option with a value other than SMALL without risking incorrect mapping of enum data types (for example, if they were used inside of a structure). For more information, see “ENUMSIZE(SMALL) and protected enumeration types in system header files” on page 92.

z/OS V1R2 introduced the ENUMSIZE option as a means for controlling the size of enumeration types. The default setting, ENUMSIZE(SMALL), provides the same behavior that occurred in previous releases of the compiler.

If you want to continue to use the ENUMSIZE option, it is recommended that the same setting be used for the whole application; otherwise, you might find inconsistencies when the same enumeration type is declared in different compilation units. Use the `#pragma enum`, if necessary, to control the size of individual enumeration types (especially in common header files).

## **Changes in compiler option functionality**

### **HALT compiler option**

As of C/C++ for MVS/ESA V3R2 compiler, the C++ compiler does not accept 33 as a valid parameter for the HALT compiler option.

### **HWOPTS compiler option**

In AD/Cycle C/370 V1, the HWOPTS compiler option directed the compiler to generate code to take advantage of different hardware. As of z/OS V1R2 C/C++ compiler, the HWOPTS compiler option is no longer supported. The replacement for it is the ARCHITECTURE option.

## INFO compiler option

As of z/OS V1R2 C/C++, the INFO option default has been changed from NOINFO to INFO(LAN) for C++.

As of z/OS V1R6 C/C++, the INFO option is supported by the C compiler as well as the C++ compiler.

**Note:** The CHECKOUT C compiler option will continue to be supported for compatibility with earlier releases only.

## INLINE compiler option

For C, the default for the INLINE compiler option was changed to 100 ACUs (Abstract Code Units) in the C/C++ for MVS/ESA compiler. Hence, with C/C++ for MVS/ESA V3R2, OS/390 C/C++, and z/OS XL C/C++ compilers, the default is 100 ACUs. In the past, the default was 250 ACUs.

For C++, the z/OS V1R1 and earlier compilers did not accept the INLINE option but did perform inlining at OPT with a fixed value of 100 for the threshold and 2000 for the limit. As of z/OS V1R2, the C++ compiler accepts the INLINE option, with defaults of 100 and 1000 for the threshold and limit, respectively. As a result of this change, code that used to be inlined may no longer be inlined due to the decrease in the limit from 2000 to 1000 ACUs.

## LANGLVL(ANSI), LANGLVL(SAA), or LANGLVL(SAAL2) compiler option and macro redefinitions

As of z/OS V1R7 XL C, the treatment of macro redefinitions has changed. For LANGLVL(ANSI), LANGLVL(SAA), or LANGLVL(SAAL2), the XL C compiler will issue a severe message instead of a warning message when a macro is redefined to a value that is different from the first definition.

```
#define COUNT 1
#define COUNT 2 /* error */
```

*Figure 5. Macro redefinition*

**Note:** Compare the treatment of macro redefinitions for these LANGLVL sub-options with that for “LANGLVL(EXTENDED) compiler option and macro redefinitions.”

## LANGLVL(EXTENDED) compiler option and macro redefinitions

As of z/OS V1R7 XL C, you can redefine a macro that has not been first undefined with LANGLVL(EXTENDED).

```
#define COUNT 1
#define COUNT 2

int main () {
    return COUNT;
}
```

*Figure 6. Macro redefinition under LANGLVL(EXTENDED)*

With z/OS V1R6 C and previous C compilers, this test will return "1". As of z/OS V1R7 XL C, this test will return "2".

**Note:** Compare the treatment of macro redefinitions for LANGLVL(EXTENDED) with that for “LANGLVL(ANSI), LANGLVL(SAA), or LANGLVL(SAAL2) compiler option and macro redefinitions.”

## LOCALE compiler option

As of z/OS V1R9 XL C/C++, the `__LOCALE__` macro is defined to the name of the compile-time locale. If you specified `LOCALE(string literal)`, the compiler uses the runtime function `setlocale(LC_ALL "string literal")` to determine the name of the compile-time locale. If you do not use the `LOCALE` compiler option, the macro is undefined.

Prior to z/OS V1R9 XL C/C++, the `__LOCALE__` macro was defined to "" when the `LOCALE` option was specified without a suboption.

## OPTIMIZE optimization level mapping

As compilers are developed, the `OPTIMIZE` optimization levels are remapped.

In the IBM C/370 compilers, `OPTIMIZE` was mapped to `OPT(1)`.

In the IBM AD/Cycle C/370 compilers:

- `OPT(0)` was mapped to `NOOPT`
- `OPT` and `OPT(1)` were mapped to `OPT(1)`
- `OPT(2)` was mapped to `OPT(2)`

In the C/C++ for MVS/ESA V3R2 compiler and the OS/390 V1R1 compiler:

- `OPT(0)` was mapped to `NOOPT`
- `OPT`, `OPT(1)` and `OPT(2)` were mapped to `OPT`

In the OS/390(r) V1R2, V1R3, V2R4, and V2R5 C/C++ compilers:

- `OPT(0)` mapped to `NOOPT`
- `OPT` and `OPT(1)` mapped to `OPT(1)`
- `OPT(2)` mapped to `OPT(2)`

As of OS/390(r) V2R6 C/C++:

- `OPT(0)` maps to `NOOPT`
- `OPT`, `OPT(1)` and `OPT(2)` map to `OPT(2)`

As of z/OS V1R5 C/C++, `OPT(3)` provides the compiler's highest and most aggressive level of optimization. `OPT(3)` is recommended only when the desire for runtime improvement outweighs the concern for minimizing compilation resources.

## SEARCH and LSEARCH compiler options

Prior to C/C++ for MVS/ESA V3R2 compilers, if you used the `LSEARCH` option more than once, the compiler would only search the libraries specified for the last `LSEARCH` option.

As of C/C++ for MVS/ESA V3R2 compilers (including z/OS XL C/C++ compiler), the compiler searches all of the libraries specified for all of the `SEARCH` options, from the point of the last `NOSEARCH` option. Previously, only the libraries specified for the last `SEARCH` option were searched.

## SQL compiler option and SQL EXEC statements

For migration information about using the `SQL` compiler option, see Chapter 21, "Migration issues with earlier C/C++ applications that use DB2 Universal Database," on page 139

## TEST compiler option

As of the OS/390 C/C++ compilers, the default for the PATH suboption of the TEST option has changed from NOPATH to PATH. Also, the INLINE option is ignored when the TEST option is in effect at OPT(0), but the INLINE option is no longer ignored if OPT(1), OPT(2), or OPT(3) is in effect.

As of C/C++ MVS V3R2 compiler, the following restriction applies to the TEST compiler option: The maximum number of lines in a single source file cannot exceed 131,072. If you exceed this limit, the results from the Debug Tool and Language Environment dump services are undefined.

As of z/OS V1R6 C/C++, when using the c89/c++ utility, the **-g** flag has changed from specifying the TEST option to DEBUG(FORMAT(DWARF)). For more information, see “Debug format specification” on page 96.

**Note:** Under ILP32 only, you can use the environment variable `{_DEBUG_FORMAT}` to determine the debug format (DWARF or ISD) to which the **-g** flag option is translated. For information about this environment variable and the c89/c++ utility, refer to the c89 utility information in *z/OS XL C/C++ User's Guide, SC14-7307*.

---

## Changes that affect compiler invocations

When you invoke the compiler, you should be aware of potential problems in the following areas:

- “IPA compiler option and very large applications”
- “Customized JCL and the CXX format”
- “CBCI and CBCXI procedures in JCL” on page 30

### IPA compiler option and very large applications

As of z/OS V1R12 XL C/C++, when using the IPA compiler option to compile very large applications, you might need to increase the size of the work file associated with SYSUTIP DD in the IPA Link step. If you are linking the application in a USS environment, you can control the size of this work file with the `_CCN_IPA_WORK_SPACE` environment variable. If particularly large source files are compiled with IPA, the default size of the compile-time work files might also need to be increased. These can be modified via the `prefix_WORK_SPACE` environment variables.

### Customized JCL and the CXX format

The CBCC, CBCCL, and CBCCLG procedures, which compile C++ code, include parameter CXX when the following compilers are used:

- C/C++ for MVS/ESA V3R2
- OS/390 C/C++
- z/OS C/C++

If you have written your own JCL to compile a C++ program, you must include this parameter; otherwise, the C compiler is invoked.

When you pass options to the compiler, you must specify parameter CXX. You must use the following format to specify options:

runtime options/CXX compiler options

## CBCI and CBCXI procedures in JCL

As of z/OS V1R5 C/C++ compiler, the CBCI and CBCXI procedures contain the variable CLBPRFX. If you have any JCL that uses these procedures, you must either customize these procedures (for example, at installation time) or modify your JCL to provide a value for CLBPRFX.

---

## Changes that affect SYSLIB DD cards

If your batch job uses a SYSLIB concatenation to search for files, remove those job steps and use the SEARCH compiler option instead.

## Change in SCLBH logical record length

As of z/OS V1R2 C/C++ compiler, the logical record length for the SCLBH data sets is increased from 80 bytes to 120 bytes. Because of this change, the SYSLIB DD card (shown in Figure 7) that specifies library search paths no longer works, and must be removed from your JCL. In its place, you must use the SEARCH compiler option.

**Example:** See the following example.

```
SEARCH(//'CEE.SCEEH.+',//'CBC.SCLBH.+')
```

Using the SEARCH compiler option instead of a SYSLIB concatenation allows the C/C++ compiler to search for files based on both file name and file type.

```
//SYSLIB DD DSN=CEE.SCEEH.H,DISP=SHR  
//      DD DSN=CEE.SCEEH.SYS.H,DISP=SHR  
//      DD DSN=CBC.SCLBH.H,DISP=SHR
```

*Figure 7. Example of SYSLIB DD cards that must be removed as of z/OS V1R2 C/C++ compiler*



---

## Chapter 6. Bind-time migration issues with pre-OS/390 C/C++ programs

This information helps you understand compatibility issues related to binding or linking executable C/C++ programs from applications that predate IBM OS/390 C/C++ compiler.

The output of a prelinking, linking, or binding process depends on where the programs are stored:

- When the programs are stored in a PDS, the output is a *load module*.
- When the programs are stored in a PDSE or in UNIX System Services files, the output is a *program object*.

For more information, see "Prelinking and linking z/OS XL C/C++ programs" and "Binding z/OS XL C/C++ programs" in *z/OS XL C/C++ User's Guide*.

**Note:** The terms in these topics that are associated with linking (bind, binding, link, link-edit) refer to the process of creating an executable program from object modules.

Generally, pre-OS/390 C/C++ load modules or programs execute successfully under z/OS V2R1 without relinking. This information highlights exceptions and shows how to solve specific problems in compatibility.

**Note:** If you are not sure which libraries were used to link an executable program, see "Library release level in use."

Executable program compatibility problems requiring source changes are discussed in Chapter 4, "Source code compatibility issues with pre-OS/390 C/C++ programs," on page 19.

When you use z/OS V2R1 XL C/C++ to bind programs that were last linked as part of pre-OS/390 C/C++ applications, be aware the following information:

- "Binder invocation changes" on page 33
- "Changes due to customizations of the runtime environment" on page 33
- "Incompatibilities in external references" on page 34
- "Requirements for relinking C/370 modules that invoke Debug Tool" on page 34
- "C/370 modules with interlanguage calls (ILC)" on page 34

Also see "Common library initialization compatibility issues with C/370 modules" on page 46.

---

### Library release level in use

The `__librel()` function is a System/370 extension to SAA C. It returns the release level of the library that your program is using, in a 32-bit integer. With Language Environment services, a field containing a number that represents the library product.

The `__librel()` return value is a 32-bit integer intended to be viewed in hexadecimal format as shown in Table 5. The hexadecimal value is interpreted as `0xPVRMMMM`, where:

- The first hex digit *P* represents the product.
- The second hex digit *V* represents the version.
- The third and fourth hex digits *RR* represent the release.
- The fifth through eighth hex digits *MMMM* represent the modification level.

Table 5. Return values for the `__librel()` function

Product	librel value
C/370 V2R2	0x02020000
Language Environment V1R5	0x11050000
OS/390 V1R1 <b>Note:</b> The <code>__librel</code> return value for OS/390 V1R1, 5645-001 is the same as it is for Language Environment V1R5 runtime libraries.	0x11050000
OS/390 V1R2	0x21020000
OS/390 V1R3	0x21030000
OS/390 V2R4	0x22040000
OS/390 V2R6	0x22060000
OS/390 V2R7	0x22070000
OS/390 V2R8	0x22080000
OS/390 V2R9	0x22090000
OS/390 V2R10	0x220A0000
z/OS V1R1	0x220A0000
z/OS V1R2	0x41020000
z/OS V1R3	0x41030000
z/OS V1R4	0x41040000
z/OS V1R5	0x41050000
z/OS V1R6	0x41060000
z/OS V1R7	0x41070000
z/OS V1R8	0x41080000
z/OS V1R9	0x41090000
z/OS V1R10	0x410A0000
z/OS V1R11	0x410B0000
z/OS V1R12	0x410C0000
z/OS V1R13	0x410D0000
z/OS V2R1	0x42010000

In C/370 V2, the high-order 8 bits were used to return the version number. Now these 8 bits are divided into two fields. The first 4 bits contain the product number and the second 4 bits contain the version number.

You must modify programs that use the information returned from `__librel()`. For more information on `__librel()`, see *z/OS XL C/C++ Runtime Library Reference, SC14-7314*.

---

## Binder invocation changes

If your application behaves unexpectedly after you relink the pre-OS/390 C/C++ modules and it includes user-developed exit routines, be aware that rules of precedence have changed.

When you bind programs that were previously compiled with an OS/390 compiler and library, you should also be aware that the following migration issues could also apply to your binder invocations:

- “Namespace pollution binder errors” on page 101
- “Program modules from an earlier release” on page 101

## Impact of changes to CC EXEC invocation syntax

As of z/OS V1R2 C/C++ compiler, there are changes in the CC EXEC invocation syntax.

At customization time, your system programmer can modify the CC EXEC to accept:

- Only the original syntax (the one supported by compilers before C/C++ for MVS/ESA V3R2).
- Only the updated syntax.
- Both syntaxes.

The CC EXEC should be customized to accept only the updated syntax.

If the CC EXEC is customized to accept both the original and additional invocations, you must choose to use either the original invocations or the updated invocations. You cannot invoke the CC command by using a mixture of both syntaxes. Be aware that the original syntax does not support UNIX System Services files provided with z/OS UNIX System Services files.

Refer to the *z/OS Program Directory* for more information about installation and customization, and to the *z/OS XL C/C++ User's Guide* for more information about compiler options.

---

## Changes due to customizations of the runtime environment

Your installation of z/OS V2R1 XL C/C++ might have been customized in ways that could affect application behavior at bind-time.

### User-developed exit routines

If your application behaves unexpectedly after you relink the pre-OS/390 C/C++ modules and if it includes user-developed exit routines, be aware that rules of precedence have changed. If both CEEBXITA and IBMBXITA are present in a relinked C/370 module, CEEBXITA will have precedence over IBMBXITA.

### Abnormal termination exit routines and dump formats

With Language Environment services in a batch environment, abnormal termination exit routine CEEBDATX is automatically linked at installation time.

This change affects you if you have supplied, or need to supply, your own exit routine. The sample exit routine had been available in the sample library provided with IBM AD/Cycle LE/370 V1R3. It automatically generates a system dump (with abend code 4039) whenever an abnormal termination occurs.

You can trigger the dump by ensuring that SYSUDUMP is defined in the GO step of the JCL that you are using (for example, by including the statement SYSUDUMP DD SYSOUT=\*).

**Note:** As of C/C++ for MVS/ESA V3R2, the standard JCL procedures shipped with the compiler do not include SYSUDUMP.

If SYSUDUMP is not included in your JCL, or is defined as DUMMY, the dump will be suppressed.

---

## Incompatibilities in external references

As of z/OS V1R3 C/C++ compiler, external names (such as entry points and external references) can be up to 32,767 bytes long.

As of z/OS V1R2 C/C++ compiler, the binder imposes a limit of 1024 characters for the length of external names. Both the OS/390 C++ compiler and z/OS C++ compiler might generate mangled names that are longer than this limit. This problem is more likely to occur when using the Standard Template Library with the z/OS V1R2 C++ compiler.

If linking programs generates mangled names that exceed the limit, do one of the following actions:

- Reduce the length of the C++ class names.
- Use the **#pragma map** directive to map the long name to a short one.
- For NOXPLINK applications, use the prelinker.

---

## Requirements for relinking C/370 modules that invoke Debug Tool

If your C/370 application has any C/C++ modules that reference the C/370 library code @@CTEST, you cannot execute them under z/OS V2R1 until you:

1. Replace the @@CTEST objects, as described in “Programs that require the C370 Common Library environment” on page 38 and in “Linkage editor control statements for modules that contain calls to COBOL routines” on page 36.
2. Relink all modules that contain calls to ctest().

---

## C/370 modules with interlanguage calls (ILC)

Table 6 outlines when a relink of ILC applications is required, based on languages found in the executable program: If you have multiple languages in the executable program, then the sum of the restrictions applies. For example: if you have C, PL/I and Fortran in the executable program, then it should be relinked because Fortran needs to be relinked. Refer to *z/OS Language Environment Writing Interlanguage Communication Applications* for more information.

Table 6. Migrations that require relinking

Language	Relink required
Assembler	No

Table 6. Migrations that require relinking (continued)

Language	Relink required
PL/I	No
Fortran	Yes
COBOL	Yes <b>Note:</b> If the C/370 ILC application is built (relinked) after the PTF for APAR PN74931 is applied, no relink is required to run under z/OS V2R1. Otherwise a relink is required.

## Interlanguage calls between assembler and PL/I language modules

Programs that contain interlanguage calls to and from assembler or PL/I language modules do not need to be relinked.

## Function calls between C and Fortran modules

For applications that use Language Environment services, Fortran/C interlanguage calls were not supported prior to the Language Environment V1R5 release and C/C++ for MVS/ESA. Before you can use Fortran/C ILC applications with Language Environment V1R5 or later, you must relink all Fortran/C ILC applications that contain pre-Language Environment C or Fortran library routines.

Before you relink those applications, be aware of the following constraints:

- You can run them with z/OS V2R1 XL C/C++ compiler only after they are relinked.
- You cannot continue to run them with the C/370 library after they are relinked.

## Function calls to and from COBOL modules

The Fortran ILC rules apply to programs that contain interlanguage calls between C/370 and COBOL, unless you relink them with the C/370 V2R1 or V2R2 library that has the PTF for APAR PN74931 applied. This PTF replaces the C/370 V2R1 and V2R2 link-edit stubs so that they tolerate Language Environment service calls. After your application is relinked using the modified C/370 V2R1 or V2R2 stubs, you can run the application with any of the following runtime environments:

- C/370 V2R1 runtime library
- C/370 V2R2 runtime library
- Language Environment runtime libraries

If you run applications with interlanguage calls (ILC) to or from COBOL without applying the PTF for APAR PN74931 and then relinking the C/370 programs that contain the ILC, be aware of the following constraints:

- You can run those applications with z/OS V2R1 only after they are relinked.
- You cannot continue to run those applications with the C/370 library after they are relinked.

## Compatibility with earlier and later releases

The PTF for APAR PN74931 replaces the link-edit stubs so that they tolerate Language Environment service calls. After your application is relinked using the

modified C/370 V2, you can run the application with the C/370 V2R1 runtime library, the C/370 V2R2 runtime library, or the Language Environment runtime libraries.

Before you can relink your C/370-COBOL ILC application with Language Environment services only, you must replace the old library objects @C2CBL and @@CBL2C, as described in “Programs that require the C370 Common Library environment” on page 38 and “Linkage editor control statements for modules that contain calls to COBOL routines.” After you replace those objects, the affected modules will be executable only with Language Environment services.

### Impact of changes in packaging of language libraries

As of z/OS V1R6, Language Environment runtime libraries contain more modules than the pre-Language Environment libraries. For example, all of the Language Environment C/C++ language libraries are packaged in both SCEERUN and SCEERUN2, instead of SCEERUN only.

The impact of these packaging changes for pre-OS/390 C/C++ applications is that certain Language Environment modules can invade user-defined name spaces. If a program uses modules that are the same as those used for Language Environment module names (such as `fetch()`), you must ensure that the program link libraries are loaded before the Language Environment libraries.

### Linkage editor control statements for modules that contain calls to COBOL routines

This topic lists the linkage editor control statements required to relink modules that contain ILCs between C and COBOL, or between C and Fortran. The object modules are compatible with the Language Environment service modules; however, the ILC linkage between the applications and the library has changed. You must relink these applications using the JCL shown in Figure 8 on page 38 and the control statements that fit your requirements from the following list. The `INCLUDE SYSLIB(@@CTDLI)` is necessary only if your program will invoke IBM IMS facilities using the z/OS XL C library function `ctdli()` and if the z/OS XL C function was called from a COBOL main program.

Control statements for various combinations of ILCs and compiler options are as follows. The modules referenced by `SYSLMOD` contain the routines to be relinked.

1. C `main()` statically calling COBOL routine B1 or dynamically calling the COBOL routine through the use of `fetch()`, where B1 was compiled with the `RES` option. Relink the C module:

```
MODE      AMODE(31),RMODE(ANY)
INCLUDE SYSLIB(EDCSTART)      ALWAYS NEEDED
INCLUDE SYSLIB(CEEROOTB)     ALWAYS NEEDED
INCLUDE SYSLIB(@C2CBL)       REQUIRED FOR C CALLING COBOL
INCLUDE SYSLIB(@@CTDLI)      REQUIRED FOR ILC & IMS
INCLUDE SYSLMOD(SAMP1)
ENTRY     CEESTART            MAIN ENTRY POINT
NAME      SAMP1(R)
```

2. C `main()` statically calling COBOL routine B2 or dynamically calling the COBOL routine through the use of `fetch()`, where B2 was compiled with the `NORES` option. Relink the C module:

```
MODE      AMODE(24),RMODE(24)
INCLUDE SYSLIB(EDCSTART)      ALWAYS NEEDED
INCLUDE SYSLIB(CEEROOTB)     ALWAYS NEEDED
INCLUDE SYSLIB(@C2CBL)       REQUIRED FOR C CALLING COBOL
INCLUDE SYSLIB(@@CTDLI)      REQUIRED FOR ILC & IMS
```

```

INCLUDE SYSLIB(IGZENRI)      REQUIRED FOR COBOL with NORES
INCLUDE SYSLMOD(SAMP2)
ENTRY CEESTART              MAIN ENTRY POINT
NAME SAMP2(R)

```

3. C main() fetches a C1 function that statically calls a COBOL routine B1 compiled with the RES option. Relink the C module:

```

MODE AMODE(31),RMODE(ANY)
INCLUDE SYSLIB(EDCSTART)    ALWAYS NEEDED
INCLUDE SYSLIB(CEEROOTB)   ALWAYS NEEDED
INCLUDE SYSLIB(@@C2CBL)    REQUIRED FOR C CALLING COBOL
INCLUDE SYSLIB(@@CTDLI)    REQUIRED FOR ILC & IMS
INCLUDE SYSLMOD(SAMP3)
ENTRY C1                    ENTRY POINT TO FETCHED ROUTINE
NAME SAMP3(R)

```

4. C main() fetches a C1 function that statically calls a COBOL routine B1 that is compiled with the NORES option. Relink the C module:

```

MODE AMODE(24),RMODE(24)
INCLUDE SYSLIB(EDCSTART)    ALWAYS NEEDED
INCLUDE SYSLIB(CEEROOTB)   ALWAYS NEEDED
INCLUDE SYSLIB(@@C2CBL)    REQUIRED FOR C CALLING COBOL
INCLUDE SYSLIB(@@CTDLI)    REQUIRED FOR ILC & IMS
INCLUDE SYSLIB(IGZENRI)    REQUIRED FOR COBOL with NORES
INCLUDE SYSLMOD(SAMP4)
ENTRY C1                    ENTRY POINT TO FETCHED ROUTINE
NAME SAMP4(R)

```

5. A COBOL main CBLMAIN compiled with the RES option statically or dynamically calls a C1 function. Relink the COBOL module:

```

MODE AMODE(31),RMODE(ANY)
INCLUDE SYSLIB(EDCSTART)    ALWAYS NEEDED
INCLUDE SYSLIB(CEEROOTB)   ALWAYS NEEDED
INCLUDE SYSLIB(IGZEBST)
INCLUDE SYSLIB(@@CBL2C)    REQUIRED FOR COBOL CALLING C
INCLUDE SYSLIB(@@CTDLI)    REQUIRED FOR ILC & IMS
INCLUDE SYSLMOD(SAMP5)
ENTRY CBLRTN               COBOL ENTRY POINT
NAME SAMP5(R)

```

6. A COBOL main CBLMAIN compiled with the NORES option statically or dynamically calls a C1 function. Relink the COBOL module:

```

MODE AMODE(24),RMODE(24)
INCLUDE SYSLIB(EDCSTART)    ALWAYS NEEDED
INCLUDE SYSLIB(CEEROOTB)   ALWAYS NEEDED
INCLUDE SYSLIB(IGZENRI)
INCLUDE SYSLIB(@@CBL2C)    REQUIRED FOR COBOL CALLING C
INCLUDE SYSLIB(@@CTDLI)    REQUIRED FOR ILC & IMS
INCLUDE SYSLMOD(SAMP6)
ENTRY CBLRTN               COBOL ENTRY POINT
NAME SAMP6(R)

```

7. C main() calls a Fortran routine. Relink the C module:

```

INCLUDE SYSLIB(EDCSTART)    ALWAYS NEEDED
INCLUDE SYSLIB(CEEROOTB)   ALWAYS NEEDED
INCLUDE SYSLIB(@@CTOF)     REQUIRED FOR C CALLING Fortran
INCLUDE SYSLIB(@@CTDLI)    REQUIRED FOR ILC & IMS
INCLUDE SYSLMOD(SAMP7)
ENTRY CEESTART              MAIN ENTRY POINT
NAME SAMP7(R)

```

8. A Fortran main() calls a C function. Relink the C module:

```

INCLUDE SYSLIB(EDCSTART)    ALWAYS NEEDED
INCLUDE SYSLIB(CEEROOTB)   ALWAYS NEEDED
INCLUDE SYSLIB(@@FTOC)     REQUIRED FOR Fortran CALLING C

```

```

INCLUDE SYSLIB(@@CTDLI)      REQUIRED FOR ILC & IMS
INCLUDE SYSLMOD(SAMP8)
ENTRY   CEESTART            MAIN ENTRY POINT
NAME    SAMP8(R)

```

For other related Fortran considerations, refer to *z/OS Language Environment Programming Guide*.

### Programs that require the C370 Common Library environment

Some legacy modules will require the C/370 Common Library environment unless they have been converted to use Language Environment services. These incompatible modules might, for example, contain ILCs to COBOL or use the library function `ctest()` to invoke the Debug Tool.

There are several methods of converting C/370 modules to use Language Environment services.

These methods are:

- Link from the original objects, using Language Environment services. The EDCSTART and CEEROOTB modules must be explicitly included.
- Relink the C/370 program, using the Language Environment CSECT replacement. The EDCSTART and CEEROOTB modules must be explicitly included.

Figure 8 shows an example of a job that uses this method. The job converts the C/370 program by relinking it and explicitly including the Language Environment CEESTART module, so that it replaces the C/370 CEESTART module.

This is a general-purpose job. The comments show the other include statements that are necessary if certain calls are present in the code. Refer to “Linkage editor control statements for modules that contain calls to COBOL routines” on page 36 for the specific control statements that are necessary for different kinds of ILCs with COBOL.

```

//Jobcard information
//*
//*****//
//*RELINK C/370 V2 USER MODULE FOR Language Environment      *//
//*****//
//*
//*
//LINK      EXEC PGM=HEWL,PARM='RMODE=ANY,AMODE=31,MAP,LIST'
//SYSPRINT DD SYSOUT=*
//SYSLIB   DD DSN=CEE.SCEELKED,DISP=SHR
//SYSLMOD  DD DSN=TSUSER1.A.LOAD,DISP=SHR
//SYSUT1   DD UNIT=VIO,SPACE=(CYL,(10,10))
//SYSLIN   DD *
           INCLUDE SYSLIB(EDCSTART)    ALWAYS NEEDED
           INCLUDE SYSLIB(CEEROOTB)   ALWAYS NEEDED
           INCLUDE SYSLIB(@@CTEST)    NEEDED ONLY IF CTEST CALLS ARE PRESENT
           INCLUDE SYSLIB(@@C2CBL)    NEEDED ONLY IF CALLS ARE MADE TO COBOL
           INCLUDE SYSLIB(@@CBL2C)    NEEDED ONLY IF CALLS ARE MADE FROM COBOL
           INCLUDE SYSLMOD(HELLO)
           ENTRY   CEESTART
           NAME    HELLO(R)
/*

```

Figure 8. Link job for converting programs

- For modules that have a C `main()` procedure:
  1. Replace the C/370 program by recompiling the source (if available).



2. Recompile the source containing the `main()` procedure with the z/OS V2R1 XL C/C++ compiler.
3. Relink the objects with Language Environment services.

**Note:** This ensures that CEESTART uses the Language Environment initialization scheme. This is an alternative to including EDCSTART explicitly when linking from objects.



---

## Chapter 7. Runtime migration issues with pre-OS/390 C/C++ applications

When you use IBM z/OS V2R1 XL C/C++ to run applications that were most recently executed prior to IBM OS/390 C/C++ compilers, be aware of the following migration issues:

- “Retention of pre-OS/390 runtime behavior”
- “Runtime library messages”
- “Changes that affect customized JCL procedures” on page 43
- “Changes in runtime option specification” on page 43
- “Runtime library compatibility issues with pre-OS/390 applications” on page 45
- “Hardware and OS exceptions” on page 48
- “Resource allocation and memory management migration issues” on page 49

---

### Retention of pre-OS/390 runtime behavior

When your program is using Language Environment services, you can use the ENVAR runtime option to specify the values of environment variables at execution time. You can use some environment variables to specify the original runtime behavior for particular items. The following setting specifies the original runtime behavior for the greatest number of items:

```
ENVAR("_EDC_COMPAT=32767")
```

Alternatively, you can add a call to the `setenv()` function, either in the CEEBINT High-Level Language exit routine or in your `main()` program. If you use CEEBINT only, you will need to relink your application. If you add a call to `setenv()` in the `main()` function, you must recompile the program and then relink your application. For more information, refer to `setenv()` in *z/OS XL C/C++ Runtime Library Reference, SC14-7314* and to Using environment variables in *z/OS XL C/C++ Programming Guide*.

---

### Runtime library messages

There are differences between pre-OS/390 and Language Environment runtime messages. Some messages have been added and some have been deleted; the contents of others have been changed. Any application that is affected by the format or contents of these messages must be updated accordingly.

**Note:** Well-formed code should not depend on message contents or message numbers.

Refer to *z/OS Language Environment Debugging Guide* for details on runtime messages and return codes.

### Return codes and messages

Since C/370 V2, library return codes and messages have been changed. Either JCL, CLISTS and EXECs that are affected by them must be changed accordingly or the CEEBXITA exit routine must be customized to emulate the old return codes. C/370

V2 return codes ranged from 0 to 999 but the Language Environment return codes have a different range. Refer to *z/OS XL C/C++ Messages, GC14-7305* for more information.

**Examples:** See the following examples.

- Return codes greater than 4095 are returned as modulo 4095 return codes.
- The return code for an abort is now 2000; it was 1000.
- The return code for an unhandled SIGFPE, SIGILL, or SIGSEGV condition is now 3000; it was 2000.

For detailed information, refer to *z/OS Language Environment Debugging Guide*.

## Error conditions that cause runtime messages

In C/370 V2, if an error was detected with the parameters being passed to the main program, the program terminated with a return code of 8 and a message indicating the reason why the program was not run. For example, if there was an error in the redirection parameters, the message would indicate that the program had terminated because of a redirection error.

Under z/OS V2R1 XL C/C++ compiler, the same message will be displayed, but the program will also terminate with a 4093 abend, reason code 52 (x'34'). For more information about reason codes see *z/OS Language Environment Debugging Guide*.

## Prefixes of perror() and strerror() messages

All Language Environment `perror()` and `strerror()` messages in C contain a prefix. (In C/370 V2, there were no prefixes to these messages.) The prefix is `EDCxxxxa`, where `xxxx` is a number (always `5xxx`) and the `a` is either I, E, or S. See *z/OS Language Environment Runtime Messages* for a list of these messages.

## Language specification for messages

Instead of specifying a messages data set for the `SYSMSG` ddname, you must now use the `NATLANG` runtime option. If you specify a data set for the `SYSMSG` ddname, it will be ignored.

**Note:** For information about the `NATLANG` runtime option, see *z/OS Language Environment Customization* and the *z/OS Language Environment Programming Reference*.

## User-developed exit routines

With Language Environment services in a batch environment, abnormal termination exit routine `CEEBDATX` is automatically linked at installation time.

This change affects you if you have supplied, or need to supply, your own exit routine. The sample exit routine had been available in the sample library provided with IBM AD/Cycle LE/370 V1R3. It automatically generates a system dump (with abend code 4039) whenever an abnormal termination occurs.

You can trigger the dump by ensuring that `SYSUDUMP` is defined in the `GO` step of the JCL that you are using (for example, by including the statement `SYSUDUMP DD SYSOUT=*`).

**Note:** As of C/C++ for MVS/ESA V3R2, the standard JCL procedures shipped with the compiler do not include `SYSUDUMP`.

If SYSUDUMP is not included in your JCL, or is defined as DUMMY, the dump will be suppressed.

---

## Changes that affect customized JCL procedures

This topic describes changes that may affect your JCL procedures, CLISTs and EXECs.

### Changes in data set names

The names of IBM-supplied data sets may change from one release to another. see *z/OS Program Directory* for more information on data set names.

### Arguments that contain a slash

You must prefix the arguments with a slash if you use Language Environment services and:

- There are no runtime options.
- The input arguments passed to `main()` contain a slash.

JCL, CLISTs, and EXECs that are affected must be changed accordingly.

### Differences in standard streams

There is no automatic association of Language Environment ddnames SYSTEM, SYSERR, SYSPRINT with the `stderr` function. In batch processes, you must use command line redirection of the type `1>&2` if you want `stderr` and `stdout` to share a device.

In C/370 V2, you could override the destination of error messages by redirecting `stderr`. The destination of all Language Environment messages is determined by the MSGFILE runtime option. See the topic on the MSGFILE runtime option in the *z/OS Language Environment Programming Guide* for more information.

### Dump generation

You can generate a dump by ensuring that SYSUDUMP is defined in the GO step of the JCL that you are using (for example, by including the statement `SYSUDUMP DD SYSOUT=*`). If SYSUDUMP is not included in your JCL, or is defined as DUMMY, the dump will be suppressed. As of C/C++ for MVS/ESA V3 compiler, the standard JCL procedures shipped with the compiler do not include SYSUDUMP.

---

## Changes in runtime option specification

This topic describes changes that might affect your specification of runtime options. For information about using pragmas in your source code to specify runtime options, see “The `#pragma runopts` directive” on page 19.

### Runtime options lists

When passing only runtime options to a C/370 V2 program, you did not have to end the arguments with a slash (/). When passing runtime options to a Language Environment program, you must end the arguments with a slash.

### Obsolete runtime options

The C/370 runtime options are mapped to Language Environment equivalents. However, if you do not use the Language Environment options, during execution

you will get a warning message which cannot be suppressed. JCL, CLISTS and EXECs that are affected by these differences must be changed accordingly.

Use the Language Environment equivalent for the C/370 V2 runtime options on the command line and in **#pragma runopts**.

ISASIZE/ISAINC	becomes	STACK
LANGUAGE	becomes	NATLANG
REPORT	becomes	RPTSTG
SPIE/STAE	becomes	TRAP
NONIPTSTACK NONONIPTSTACK	becomes	XPLINK

## Return codes for abnormal enclave terminations

As of OS/390 V2R9, the default option for ABTERMENC is ABEND instead of RETCODE. If your program depends on the default behavior of ABTERMENC to be RETCODE, you must change the setting in CEEDOPT (CEECOPT for CICS). For details about changing CEEDOPT and CEECOPT, refer to *z/OS Language Environment Customization, SA38-0685*.

## Abnormal terminations and the TRAP runtime option

STAE and SPIE runtime options have been replaced with the TRAP runtime option. IBM recommends that you use the TRAP(ON,SPIE) option, not STAE and SPIE. However, for ease of migration, the STAE and SPIE options are supported *as long as the TRAP option is not explicitly specified*.

TRAP(ON) must be in effect for the ABTERMENC runtime option to have effect. For more information, refer to ABTERMENC and TRAP in *z/OS Language Environment Programming Reference*.

## Default heap allocations

The default size and increment for Language Environment HEAP runtime option differ from those of the C/370 V2 HEAP runtime option. The C/370 V2 defaults were 4K size and 4K increment.

The Language Environment defaults are:

- For CICS applications: HEAP(32K,32K,ANYWHERE,KEEP,8K,4K)
- For non-CICS applications: HEAP(4K,4080,ANYWHERE,KEEP,4K,4080)

The amount of heap storage allocated and incremented below the 16M line is determined by the following Language Environment parameters:

- initsz24.
- incrsz24.

For information about these parameters, see *z/OS Language Environment Programming Reference*.

## HEAP parameter specification

In IBM C/370 V2, only the first two of the four parameters for the HEAP option were positional. The keyword parameters could be specified if the first two were omitted. All Language Environment parameters are positional. To specify the KEEP parameter only, you must enter HEAP(,,KEEP).

## Default stack allocations

The Language Environment STACK option defaults for size and increment differ from the defaults in C/370 V2, which were 0K size and 0K increment.

Language Environment STACK option defaults are:

- For non-CICS, non-XPLINK applications:  
STACK(128K,128K,ANYWHERE,KEEP,512K,128K)
- For non-CICS, XPLINK applications:  
STACK(512K,128K,ANYWHERE,KEEP,512K,128K)
- For CICS, non-XPLINK applications:  
STACK(4K,4080,ANYWHERE,KEEP,4K,4080)
- For CICS, XPLINK applications: STACK(4K,4080,ANYWHERE,KEEP,4K,4080)

## STACK parameter specification

All Language Environment STACK parameters are positional. In other words, the keyword parameter could be specified if the first two were omitted. To specify only ANYWHERE you must enter: STACK(,ANYWHERE).

**Note:** In C/370 V2, only the first two parameters were positional.

## XPLINK downward-growing stack and the THREADSTACK runtime option

In OS/390 V2R10, the THREADSTACK runtime option replaced the NONIPTSTACK and NONONIPTSTACK options. The OS/390 V2R10 options are still accepted, but an information message will be issued, telling you to switch to the THREADSTACK option.

Be aware that the OS/390 V2R10 options do not support specification of the initial and increment sizes of the XPLINK downward-growing stack. For more information about the THREADSTACK runtime option, refer to *z/OS Language Environment Customization, SA38-0685*.

---

## Runtime library compatibility issues with pre-OS/390 applications

Changes in runtime libraries might cause problems when you run pre-OS/390 C/C++ applications. Be aware of the following issues:

- “Changes to the putenv() function and POSIX compliance”
- “UCMAPS and UCS-2 and UTF-8 converters” on page 46
- “Common library initialization compatibility issues with C/370 modules” on page 46
- “Internationalization issues in POSIX and non-POSIX applications” on page 47

## Changes to the putenv() function and POSIX compliance

As of z/OS V1R5 C/C++, the function putenv() places the string passed to putenv() directly into the array of environment variables. This behavior assures compliance with the POSIX standard.

Prior to z/OS V1R5 C/C++, the string used to define the environment variable passed into putenv() was not added to the array of environment variables. Instead, the system copied the string into system-allocated storage.

To allow the POSIX-compliant behavior of `putenv()`, do nothing; it's now the default condition.

To restore the previous behavior of `putenv()`, follow these steps:

1. Ensure that the environment variable, `_EDC_PUTENV_COPY`, is available on your pre-z/OS V1R5 system.
2. Set the environment variable `_EDC_PUTENV_COPY` to "YES".

For additional information, see:

- *z/OS XL C/C++ Runtime Library Reference*
- `_EDC_PUTENV_COPY` in *z/OS XL C/C++ Programming Guide*

## UCMAPS and UCS-2 and UTF-8 converters

As of OS/390 V2R9, the compiler supported direct use of the UCS-2 and UTF-8 converters; the tables generated by the processing of UCMAPS by the `uconvdef` utility are no longer used. This is a migration issue if you modified UCMAPS to use the UCS-2 and UTF-8 converters. If you still need to use the modifications that you made to UCMAPS, you will now need to set the `_ICONV_UCS2` environment variable to "O". For more information about the `_ICONV_UCS2` environment variable, refer to *z/OS XL C/C++ Programming Guide, SC14-7315*.

## Common library initialization compatibility issues with C/370 modules

Both Language Environment modules and C/370 modules use static code and dynamic code. Static code sections are emitted or bound with the main program object. Dynamic code sections are loaded and executed by the static component.

The sequence of events during initialization for C/370 modules differs from that for Language Environment modules. The key static code for the CEESTART object controls initialization at execution time. The C/370 CEESTART object contents differ from those of the Language Environment CEESTART object. Its contents differ between the products. The Language Environment key dynamic code for the CEESTART object is CEEBINIT, which is stored in SCEERUN. The C/370 R2 key dynamic code for the CEESTART object is IBMBLIIA, which is a Common Library part stored in SIBMLINK. The Common Library is used by the C/370 V2 libraries.

### Initialization schemes

The tables in this topic describe the initialization schemes for the CEESTART and IBMBLIIA modules:

- Table 7 describes the initialization scheme for C/370 V2 modules.
- Table 8 on page 47 describes the initialization scheme for Language Environment modules.
- Table 9 on page 47 describes the Language Environment initialization scheme for C/370 programs.

The following describes the C/370 V2 initialization scheme:

*Table 7. C/370 V2 initialization scheme*

Stage	Description
Load	The C/370 V2 CEESTART loads IBMBLIIA.
Initialize	IBMBLIIA initializes the Common Library.
Run	The Common Library runs C/370-specific initialization.



Table 7. C/370 V2 initialization scheme (continued)

Stage	Description
Call	The main program is called.

The following describes the initialization scheme:

Table 8. Language Environment initialization scheme

Stage	Description
Load	CEESTART loads CEEBINIT.
Initialize	CEEBINIT initializes Language Environment services.
Run	The Language Environment runtime library runs the C-specific initialization.
Call	The main program is called.

Table 9. Language Environment initialization scheme for C/370 programs

Stage	Description
Load	C/370 V2 CEESTART loads CEEBLIIA (as IBMBLIIA).
Initialize	CEEBLIIA (IBMBLIIA) initializes Language Environment services.
Run	The Language Environment runtime library runs the C-specific initialization.
Call	The main program is called.

In Table 9, compatibility with C/370 V2 programs depends upon the program's ability to intercept the initialization sequence at the start of the dynamic code and to initialize the Language Environment services at that point. This interception is achieved by the addition of a part named CEEBLIIA, which has been assigned the alias IBMBLIIA. This provides "initialization compatibility".

### Special considerations: CEEBLIIA and IBMBLIIA

The only way to control which environment is initialized for a given C/370 V2 program (when CEEBLIIA is assigned the alias of IBMBLIIA) is to correctly arrange the concatenation of libraries.

The version of IBMBLIIA that is found first determines the services (Language Environment or Common Library) that are initialized.

- If you intend to initialize the Common Library services, ensure that SIBMLINK is concatenated before SCEERUN.
- If you intend to initialize the Language Environment services, ensure that SCEERUN is concatenated before SIBMLINK.

## Internationalization issues in POSIX and non-POSIX applications

You should customize your locale information. Otherwise, in rare cases, you may encounter errors. In a POSIX application, you can supply time zone and alternative time (for example, daylight) information with the TZ environment variable. In a non-POSIX application, you can supply this information with the \_TZ environment variable. If no \_TZ environment variable is defined for a POSIX application or no \_TZ environment variable is defined for a non-POSIX application, any customized information provided by the LC\_TOD locale category is used. By setting the TZ

environment variable for a POSIX application, or the `_TZ` environment variable for a non-POSIX application, or by providing customized time zone or daylight information in an `LC_TOD` locale category, you allow the time functions to preserve both time and date, correctly adjusting for alternative time on a given date.

Refer to *z/OS XL C/C++ Programming Guide* for more information about both environment variables and customizing a locale.

---

## Hardware and OS exceptions

The following points identify migration and coexistence considerations for user applications:

- CICS programs that use Language Environment services are enabled for decimal overflow exceptions.
- The C packed-decimal support routines are not supported in an environment that exploits asynchronous events.

### Decimal overflow exceptions

Language Environment services support the packed decimal overflow exception using IBM System zArchitecture systems.

The value of the program mask in the program status word (PSW) is 4 (decimal overflow enabled). See “Unexpected SIGFPE exceptions” and “Explicit program mask manipulations” on page 22.

### SIGTERM, SIGINT, SIGUSR1, and SIGUSR2 exceptions

SIGTERM, SIGINT, SIGUSR1, and SIGUSR2 exceptions are handled differently for C/370 V2 and Language Environment programs.

The differences or incompatibilities are:

- The defaults for the SIGINT, SIGTERM, SIGUSR1, and SIGUSR2 signals changed in AD/Cycle LE/370 V1R3 from what they were in C/370 V1 and V2 and AD/Cycle LE/370 V1R1 and V2R2. These changes were carried into the Language Environment runtime environment. In the C/370 library and AD/Cycle LE/370 V1R1 and V1R2, the defaults for SIGINT, SIGUSR1, and SIGUSR2 were to ignore the signals. As of AD/Cycle LE/370 V1R3, the defaults are to terminate the program and issue a return code of 3000. For SIGTERM, the default has always been to terminate the program. The return code is "3000"; before, it was "0".
- Language Environment programs that terminate abnormally will not drive the `atexit` list.

### Unexpected SIGFPE exceptions

Decimal overflow conditions were masked in the C/370 library prior to V2R2. Diagnosis of overflow conditions were enabled when the packed decimal data type was introduced prior to C/370 V2R2.

As of z/OS V1R7 XL C/C++ compiler, load modules that had generated decimal overflow conditions might raise unexpected SIGFPE exceptions. You cannot migrate such modules to the current without altering the source.

**Note:** These unexpected exceptions are most likely to occur in mixed language modules, particularly those using C and assembler code where the assembler code explicitly manipulates the program mask. See “Explicit program mask manipulations” on page 22.

---

## Resource allocation and memory management migration issues

Incompatibilities in memory management might cause unexpected results in the output of your program. In your source code, you should be aware of potential problems when you use any operators or structures that re-allocate resources during application execution.

### The `realloc()` function

If Language Environment services are initialized when the `realloc()` function is used, a new storage area is obtained and the data is copied. Under C/370 V2, the `realloc()` function will reuse an area unless the function needs a larger area.

If your program uses Language Environment services, ensure that the source code does not depend on the C/370 V2 behavior of the `realloc()` function.



---

## Chapter 8. Input and output operations compatibility

Language Environment V1R5 input and output support differs from that provided by pre-OS/390 libraries. If your programs last performed input and output operations with a pre-OS/390 C/C++ compiler, you should read the changes listed herein.

**Note:** In this information, references to "previous releases" or "previous behavior" apply either to pre-OS/390 compilers or to a runtime environment that precedes the Language Environment V1R5 release.

You will generally be able to migrate "well-behaved" programs: programs that do not rely on undocumented behavior, restrictions, or invalid behaviors of previous releases. For example, if library documentation specified only that a return code was a negative value, and your code relies on that value being "-3", your code is not well-behaved and is relying on undocumented behavior.

Another example of a program that is not well-behaved is one that specifies `recfm=F` for a terminal file and depends on the runtime environment to ignore this parameter, as it did previously.

You might need to change even well-behaved code under circumstances described in the following topics.

---

### Migration issues when opening pre-OS/390 files

When you call the `fopen()` or `freopen()` library function, you can specify each parameter only once. If you specify any keyword parameter in the *mode* string more than once, the function call fails. Previously, you could specify more than one instance of a parameter.

The library no longer supports uppercase open modes on calls to `fopen()` or `freopen()`. You must specify, for example, `rb` instead of `RB`, to conform to the ANSI/ISO standard.

You cannot open a non-HFS file more than once for a write operation. Previous releases allowed you, in some cases, to open a file for write more than once. For example, you could open a file by its data set name and then again by its `ddname`. This is no longer possible for non-HFS files, and is not supported.

Previously, `fopen()` allowed spaces and commas as delimiters for mode string parameters. Only commas are allowed now.

If you are using PDSs or PDSEs, you cannot specify any spaces before the member name.

---

### Migration issues when writing to pre-OS/390 files

Write operations to files opened in binary mode are no longer deferred. Previously, the library did not write a block that held *nm* bytes out to the system until the user wrote *nm+1* bytes to the block. Language Environment services follow the rules for full buffering, described in *z/OS XL C/C++ Programming Guide*, and write data as

soon as the block is full. The *nm* bytes are still written to the file, the only difference is in the timing of when it is done.

For non-terminal files, the backspace character ('`\b`') is now placed into files as is. Previously, it backed up the file position to the beginning of the line.

For all text I/O, truncation for `fwrite()` is now handled the same way that it is handled for `puts()` and `fputs()`. If you write more data than a record can hold, and your output data contains any of the terminating control characters, '`\n`' or '`\r`' (or '`\f`', if you are using ASA), the library still truncates extra data; however, recognizing that the text line is complete, the library writes subsequent data to the next record boundary. Previously, `fwrite()` stopped immediately after the library began truncating data, so that you had to add a control character before writing any more data.

You can now partially update a record in a file opened with `type=record`. Previous services returned an error if you tried to make a partial update to a record. Now, a record is updated up to the number of characters you specify, and the remaining characters are untouched. The next update is to the next record.

Language Environment services block files more efficiently than some previous services did. Applications that depend on the creation of short blocks may fail.

The behavior of ASA files when you close them has changed. In previous releases, this is what happened:

Written to file	Read from file after <code>fclose()</code> , <code>fopen()</code>
<code>abc\n\n\n</code>	<code>abc\n\n\n\n</code>
<code>abc\n\n</code>	<code>abc\n\n\n</code>
<code>abc\n</code>	<code>abc\n</code>

Starting with this release, you read from the file what you wrote to it. For example:

Written to file	Read from file after <code>fclose()</code> , <code>fopen()</code>
<code>abc\n\n\n</code>	<code>abc\n\n\n</code>
<code>abc\n\n</code>	<code>abc\n\n</code>
<code>abc\n</code>	<code>abc\n</code>

With previous services, writing a single new-line character to a new file created an empty file under MVS™. Language Environment services treat a single new-line character written to a new file as a special case, because it is the last new-line character of the file. A single blank is written to the file. When this file is read, there are two new-line characters instead of one. There are also two new-line characters if two new-line characters were written to the file.

The behavior of appending to ASA files has also changed. The following table shows what you get from an ASA file when you:

1. Open an ASA file for write.
2. Write `abc`.
3. Close the file.
4. Append `xyz` to the ASA file.
5. Open the same ASA file for read.

Table 10. Appending to ASA files

abc Written to file, fclose() then append xyz	What you read from file after fclose(), fopen()	
	Previous release	New release
abc ==> xyz	\nabc\nxyz\n	same as previous release
abc ==> \nxyz	\nabc\nxyz\n	\nabc\n\nxyz\n
abc ==> \rxyz	\nabc\rxyz\n	\nabc\n\rxyz\n
abc\n ==> xyz	\nabc\nxyz\n	same as previous release
abc\n ==> \nxyz	\nabc\nxyz\n	\nabc\n\nxyz\n
abc\n ==> \rxyz	\nabc\rxyz\n	\nabc\n\rxyz\n
abc\n\n ==> xyz	\nabc\n\n\nxyz\n	\nabc\n\nxyz\n
abc\n\n ==> \nxyz	\nabc\n\n\nxyz\n	same as previous release
abc\n\n ==> \rxyz	\nabc\n\n\nrxyz\n	same as previous release

## Changes in DBCS string behavior

I/O now checks the value of `MB_CUR_MAX` to determine whether to interpret DBCS characters within a file.

When `MB_CUR_MAX` is 4, you can no longer place control characters in the middle of output DBCS strings for interpretation. Control characters within DBCS strings are treated as DBCS data. This is true for terminals as well. Previous products split the DBCS string at the '\n' (new-line) control character position by adding an SI (Shift In) control character at the new-line position, displaying the line on the terminal, and then adding an SO (Shift Out) control character before the data following the new-line character. If `MB_CUR_MAX` is 1, the library interprets control characters within any string, but does not interpret DBCS strings. SO and SI characters are treated as ordinary characters.

When you are writing DBCS data to text files, if there are multiple SO (Shift Out) control-character write operations with no intervening SI (Shift In) control character, the library discards the SO characters, and marks that a truncation error has occurred. Previous products allowed multiple SO control-character write operations with no intervening SI control character without issuing an error condition.

When you are writing DBCS data to text files and specify an odd number of DBCS bytes before an SI control character, the last DBCS character is padded with a 'X'FE' byte. If a SIGIOERR handler exists, it is triggered. Previous products allowed incorrectly placed SI control-character write operations to complete without any indication of an error.

Now, when an SO has been issued to indicate the beginning of a DBCS string within a text file, the DBCS must terminate within the record. The record will have both an SO and an SI.

## Changes in stdout and stderr file positioning

The Language Environment inheritance model for standard streams supports repositioning. Previously, if you opened `stdout` or `stderr` in update mode, and then called another C program by using the ANSI-style `system()` function, the program that you called inherited the standard streams, but moved the file

position for stdout or stderr to the end of the file. Now, the library does not move the file position to the end of the file. For text files, the position is moved only to the nearest record boundary not before the current position. This is consistent with the way stdin behaves for text files.

The values for `L_tmpnam` and `FILENAME_MAX` have been changed:

Constant	Old values	New values
<code>L_tmpnam</code>	47	1024
<code>FILENAME_MAX</code>	57	1024

The names produced by the `tmpnam()` library function are now different. Any code that depends on the internal structure of these names may fail.

The behavior of `fgetpos()`, `fseek()` and `fflush()` following a call to `ungetc()` has changed. Previously, these functions have all ignored characters pushed back by `ungetc()` and have considered the file to be at the position where the first `ungetc()` character was pushed back. Also, `ftell()` acknowledged characters pushed back by `ungetc()` by backing up one position if there was a character pushed back.

Now:

- `fgetpos()` behaves just as `ftell()` does.
- When a seek from the current position (`SEEK_CUR`) is performed, `fseek()` accounts for any `ungetc()` character before moving, using the user-supplied offset.
- `fflush()` moves the position back one character for every character that was pushed back.

If you have applications that depend on the previous behavior of `fgetpos()`, `fseek()`, or `fflush()`, you may use the `_EDC_COMPAT` environment variable so that source code need not change to compensate for the change in behavior.

For OS I/O to and from files opened in text mode, the `ftell()` encoding system now supports higher blocking factors for smaller block sizes. In general, you should not rely on `ftell()` values generated by code you developed using previous releases of the library. You can try `ftell()` values taken in previous releases for files opened in text or binary format if you set the environment variable `_EDC_COMPAT` before you call `fopen()` or `freopen()`. Do not rely on `ftell()` values saved across program boundaries.

For record I/O, `ftell()` now returns the relative record number instead of an encoded offset from the beginning of the file. You can supply the relative record number without acquiring it from `ftell()`. You cannot use old `ftell()` values for record I/O, regardless of the setting of `_EDC_COMPAT`.

After you have called `ftell()`, calls to `setbuf()` or `setvbuf()` might fail. Applications should never call I/O functions between calls to `fopen()` or `freopen()` and calls to the functions that control buffering.

**Note:** `_EDC_COMPAT` is described in *z/OS XL C/C++ Programming Guide*.



## Behavior changes when closing and reopening ASA files

The behavior of ASA files when you close and reopen them is now consistent: For more information about using ASA files, refer to *z/OS XL C/C++ Programming Guide*.

Table 11. Closing and reopening ASA files

Written to file	Physical record after close					
	Previous behavior			New behavior		
abc	Char	abc	(1)	same as previous release		
	Hex	4888 0123	(1)			
abc\n	Char	abc	(1)	same as previous release		
	Hex	4888 0123	(1)			
abc\n\n	Char	abc	(1)	Char	abc	(1)
		0	(2)			
	Hex	4888 0123	(1)	Hex	4888 0123	(1)
		F 0	(2)			
abc\n\n\n	Char	abc	(1)	Char	abc	(1)
		-	(2)			
	Hex	4888 0123	(1)	Hex	4888 0123	(1)
		6 0	(2)			
abc\r	Char	abc	(1)	same as previous release		
		+	(2)			
	Hex	4888 0123 4 E	(1) (2)			
abc\f	Char	abc	(1)	same as previous release		
		1	(2)			
	Hex	4888 0123 F 1	(1) (2)			

## Changes in values returned by the fldata() function

There are minor changes to the values returned by the fldata() library function. It may now return more specific information in some fields. For more information, refer to "fldata() behavior", in *z/OS XL C/C++ Programming Guide*.

## VSAM I/O changes

- The library no longer appends an index key when you read from an RRDS file opened in text or binary mode.
- RRDS files opened in text or binary mode no longer support setting the access direction to BWD.

## Change in allocation of VSAM control blocks and I/O buffers

As of z/OS V1R10, the XL C/C++ compiler instructs VSAM, by default, to allocate control blocks and I/O buffers above the 16-MB line.

If you determine that this change could be causing a problem, you can use the VSAM JCL parameter AMP to override the default.

---

### Terminal I/O changes

The library will now use the actual `recfm` and `lrecl` specified in the `fopen()` or `freopen()` call that opens a terminal file. Incomplete new records in fixed binary and record files are padded with blank characters until they are full, and the `__recfmF` flag is set in the `fldata()` structure. Previously, MVS terminals unconditionally set `recfm=U`. Terminal I/O did not support opening files in fixed format.

The use of an `LRECL` value in the `fopen()` or `freopen()` call that opens a file sets the record length to the value specified. Previous releases unconditionally set the record length to the default values.

For input text terminals, an input record now has an implicit logical record boundary at `LRECL` if the size of the record exceeds `LRECL`. The character data in excess of `LRECL` is discarded, and a `'\n'` (new-line) character is added at the end of the record boundary. You can now explicitly set the record length of a file as a parameter on the `fopen()` call. The old behavior was to allow input text records to span multiple `LRECL` blocks.

Binary and record input terminals now flag an end-of-file condition with an empty input record. You can clear the EOF condition by using the `rewind()` or `clearerr()` library function. Previous products did not allow these terminal types to signal an end-of-file condition. The use of a `RECFM` value in the `fopen()` or `freopen()` call that opens a file sets the record format to the value specified. Previous releases unconditionally set the record format to the default values.

When an input terminal requires input from the system, all output terminals with unwritten data are flushed in a way that groups the data from the different open terminals together, each separated from the other with a single blank character. The old behavior is equivalent to the new behavior, except that two blank characters separate the data from each output terminal.

---

## Part 3. Migration of OS/390 C/C++ applications to z/OS V2R1 XL C/C++

OS/390 C/C++ applications were created with one of the following products:

- IBM OS/390 V1R1 C/C++ (reship of IBM C/C++ for MVS/ESA V3R2)
- IBM OS/390 V1R2 or V1R3 C/C++
- IBM OS/390 V2R4, V2R5, V2R6, V2R7, V2R8, V2R9, or V2R10 C/C++
- IBM z/OS V1R1 C/C++ (reship of IBM OS/390 V2R10 C/C++)

### Notes:

1. The z/OS V1R1 compiler and library are equivalent to the OS/390 V2R10 compiler and library.
2. The OS/390 V2R5 compiler is equivalent to the OS/390 V2R4 compiler.
3. The OS/390 V1R1 compiler and library are equivalent to the final MVS/ESA compiler and library, and are described in Part 2, "Migration of pre-OS/390 C/C++ applications to z/OS V2R1 XL C/C++," on page 17.

Generally, you can bind OS/390 programs successfully with z/OS V2R1 programs without changing source code, and without recompiling or relinking programs.

The following topics provide information relevant to migrating a OS/390 application to z/OS V2R1 XL C/C++:

- Chapter 9, "Source code compatibility issues with OS/390 programs," on page 59
- Chapter 10, "Compile-time migration issues with OS/390 programs," on page 61
- Chapter 11, "Bind-time migration issues with OS/390 C/C++ programs," on page 71
- Chapter 12, "Runtime migration issues with OS/390 C/C++ applications," on page 73
- Chapter 13, "Migration issues resulting from class library changes between OS/390 C/C++ applications and Standard C++ library," on page 75

### Notes:

1.  
If your application uses IBM CICS information or statements, also see Chapter 20, "Migration issues with earlier C/C++ applications that run CICS statements," on page 133.
2.  
If your application uses IBM DB2 information or statements, also see Chapter 21, "Migration issues with earlier C/C++ applications that use DB2 Universal Database," on page 139.



---

## Chapter 9. Source code compatibility issues with OS/390 programs

In general, you can use source programs with the z/OS V2R1 XL C/C++ compiler without modification, if they were created with an OS/390 compiler and library.

For details on support of *Programming languages - C++ (ISO/IEC 14882:2003(E))*, see Part 5, “ISO Standard C++ compliance migration issues,” on page 115.

**Note:** Some source code compatibility issues can be addressed by modifying runtime options. See Chapter 12, “Runtime migration issues with OS/390 C/C++ applications,” on page 73.

---

### Overflow processing and code modifications

When a data type conversion causes an overflow (that is, the floating type value is larger than INT\_MAX), the behavior is undefined according to the C Standard. The actual result depends on the ARCHITECTURE level (the ARCH option), which determines the machine instruction used to do the conversion. For example, there are input values that would result in a large negative value for ARCH(2) and below, while the same input would result in a large positive value for ARCH(3) and above.

If overflow processing is important to the program, the code should provide explicit checks.

Table 12. Modifying code to check overflow processing

Example of code that does not check overflow processing	Example of code that is modified to check overflow processing
<pre>double x; int i; /* ... */  i = x; /* overflow if x is too large */ /* value of i undefined */</pre>	<pre>double x; int i; if (x &lt; (double) INT_MAX)     i = x; else {     /* overflow */ }</pre>

---

### References to class libraries that are no longer shipped

As of z/OS V1R9, IBM Open Class Library (IOC) dynamic link libraries (DLLs) are no longer shipped with the z/OS XL C/C++ compiler.

Any source dependency on an IOC DLL must be removed.

For information about the libraries that are supported by the current release, see *z/OS XL C/C++ Runtime Library Reference*.



---

## Chapter 10. Compile-time migration issues with OS/390 programs

When you compile programs that were previously compiled with an OS/390 compiler and library, be aware of the following migration issues:

- “Changes in compiler listings and messages”
- “Changes in compiler options” on page 62
- “Changes in IBM data set names” on page 69
- “Introduction of 1998 Standard C++ support” on page 69
- “Changes that affect performance and optimization” on page 69
- “Removal of Model Tool support” on page 70

---

### Changes in compiler listings and messages

From release to release, message contents can change and, for some messages, return codes can change. Errors can become warnings, and warnings can become errors. You must update any application that is affected by changes in message contents or return codes. Do not build dependencies on message contents, message numbers, or return codes. See *z/OS XL C/C++ Messages* for a list of compiler messages.

Listing formats, especially the pseudo-assembler parts, will continue to change from release to release. Do not build dependencies on the structure or content of listings. For information about C listings or the C++ listings for the current release, refer to *z/OS XL C/C++ User's Guide, SC14-7307*.

### Debug format specification

As of z/OS V1R6 C/C++, the environment variable `_DEBUG_FORMAT` can be used with the c89 utility to specify translation of the `-g` flag option for 31-bit compilations:

- If `_DEBUG_FORMAT` equals DWARF (the default), `-g` is translated to `DEBUG(FORMAT(DWARF))`.
- If `_DEBUG_FORMAT` equals ISD, then `-g` is translated to TEST (the old translation).

For the impact on the runtime environment, see “Debug format and translation of the c89 -g flag option” on page 74.

For more information about using the c89 utility, see the c89 utility information in *z/OS XL C/C++ User's Guide*.

### Language specification for compiler messages

With the C/C++ for MVS/ESA V3R2, OS/390, and z/OS XL C/C++ compilers, the method of specifying the language for compiler messages has changed. At compile time, instead of specifying message data sets on the `SYSMMSG` and `SYSMMSG` ddnames, you must now use the `NATLANG` runtime option. If you specify data sets for these ddnames, they are ignored.

**Note:** For information about the NATLANG runtime option, see *z/OS Language Environment Customization* and the *z/OS Language Environment Programming Reference*.

## Optimization level mapping and listing content

As of OS/390 V2R6 C/C++ compiler, OPT, OPT(1), and OPT(2) map to OPT(2). The compiler listing no longer contains the part of the pseudo-assembler listing that was associated with OPT(1). Listing formats, especially the pseudo-assembler parts, will continue to change from release to release. Do not build dependencies on the structure or content of listings. For information about C listings or C++ listings for the current release, refer to *z/OS XL C/C++ User's Guide*.

## Macro redefinitions and error messages

As of z/OS V1R7 XL C, the behavior of macro redefinition has changed. For certain language levels, the XL C compiler will issue a severe error message instead of a warning message when a macro is redefined to a value that is different from the first definition.

For information about the language levels that are affected, see “LANGLVL(ANSI), LANGLVL(SAA), or LANGLVL(SAAL2) compiler option and macro redefinitions” on page 66 and “LANGLVL(EXTENDED) compiler option and macro redefinitions” on page 66.

---

## Changes in compiler options

As the compiler is developed, some options are no longer supported and others undergo functional changes, such as adjustments in the default values.

### Compiler options that are no longer supported

As of z/OS V1R2 C/C++ compiler, the following compiler options are no longer supported:

- DECK

The replacement for DECK functionality that routes output to DD:SYSPUNCH is to use OBJECT(DD:SYSPUNCH).

- GENPCH
- HWOPTS

The replacement for HWOPTS is ARCHITECTURE.

- LANGLVL(COMPAT)
- OMVS

The replacement for OMVS is OE.

- SRCMSG
- SYSLIB

The replacement for SYSLIB is SEARCH.

- SYSPATH

The replacement for SYSPATH is SEARCH.

- USEPCH
- USERLIB

The replacement for USERLIB is LSEARCH.

- USERPATH

The replacement for USERPATH is LSEARCH.



As of OS/390 V2R10 C/C++ compiler, the following SOM-related compiler options are no longer supported:

- SOM | NOSOM
- SOMEinit | NOSOMEinit
- SOMGs | NOSOMGs
- SOMRo | NOSOMRo
- SOMVolattr | NOSOMVolattr
- XSominc | NOXSominc

## ARCHITECTURE compiler option

As of z/OS V2R1 XL C/C++ compiler, the default value of the ARCHITECTURE compiler option is 7.

As of z/OS V1R6 C/C++ compiler, the default value of the ARCHITECTURE compiler option is 5.

In OS/390 V2R10 to z/OS V1R5 releases, the default value of the ARCHITECTURE compiler option is 2. In OS/390 V2R9 C/C++ and previous releases, the default value of the ARCHITECTURE compiler option is 0.

### ARCHITECTURE level and overflow processing

When a data conversion causes an overflow (for example, the floating type value is larger than INT\_MAX), the behavior is undefined according to the C Standard.

The actual result depends on the ARCHITECTURE level (the ARCH option), which determines the machine instruction used to do the conversion. For example, there are input values that would result in a large negative value for ARCH(2) and below, while the same input would result in a large positive value for ARCH(3) and above.

For more information, see “Overflow processing and code modifications” on page 59.

### ARCHITECTURE level and Metal C file-scope header SYSSTATE ARCHLVL statement

The SYSSTATE ARCHLVL statement in the Metal C file-scope header identifies the minimum hardware requirement.

Starting from z/OS V2R1 XL C++ compiler, if and only if ARCH(7) or up and OSREL(ZOSV2R1) or higher are in effect, SYSSTATE ARCHLVL=3; otherwise, SYSSTATE ARCHLVL=2.

## ARGPARSE compiler option with Metal

Starting from z/OS V1R13 XL C++ compiler, the ARGPARSE option is supported with the METAL option. For more information, see **ARGPARSE | NOARGPARSE** that is documented in *z/OS XL C/C++ User's Guide*.

## ASCII compiler option

As of z/OS V1R10 XL C++ compiler, the Unicode characters that use \U or \u notation are always sensitive to the ASCII compiler option. When the ASCII option is in effect, those characters are encoded in ASCII, even when they are found in **#pragma comment** directives. Prior to z/OS V1R10 XL C++ compiler, all **#pragma comment** text strings were encoded in EBCDIC.

## CHECKOUT(CAST) compiler option

This suboption instructs the C compiler to check the source code for pointer casting that might affect optimization (that is, for those castings that violate the ANSI-aliasing rule). For detailed information, refer to information about the ANSIALIAS option in *z/OS XL C/C++ User's Guide*.

Prior to z/OS V1R2 C/C++ compiler, the compiler issued a warning message whenever this condition was detected. As of z/OS V1R2 C/C++ compiler, this message is informational. If you want to be alerted by the compiler that this message has been issued, you can use the HALTONMSG compiler option. The HALTONMSG option causes the compiler to stop after source code analysis, skip the code generation, and issue a return code of 12.

## DIGRAPH compiler option

As of z/OS V1R2 C/C++ compiler, the DIGRAPH option default for C and C++ has been changed from NODIGRAPH to DIGRAPH.

## ENUMSIZE compiler option

As of z/OS V1R7 XL C/C++, selected enumerated (enum) type declarations in system header files are protected to avoid potential execution errors. This allows you to specify the ENUMSIZE compiler option with a value other than SMALL without risking incorrect mapping of enum data types (for example, if they were used inside of a structure). For more information, see “ENUMSIZE(SMALL) and protected enumeration types in system header files” on page 92.

z/OS V1R2 introduced the ENUMSIZE option as a means for controlling the size of enumeration types. The default setting, ENUMSIZE(SMALL), provides the same behavior that occurred in previous releases of the compiler.

If you want to continue to use the ENUMSIZE option, it is recommended that the same setting be used for the whole application; otherwise, you might find inconsistencies when the same enumeration type is declared in different compilation units. Use the `#pragma enum`, if necessary, to control the size of individual enumeration types (especially in common header files).

## INFO compiler option

As of z/OS V1R2 C/C++, the INFO option default has been changed from NOINFO to INFO(LAN) for C++.

As of z/OS V1R6 C/C++, the INFO option is supported by the C compiler as well as the C++ compiler.

**Note:** The CHECKOUT C compiler option will continue to be supported for compatibility with earlier releases only.

## INLINE compiler option

For C++, the z/OS V1R1 and earlier compilers did not allow you to change the inlining threshold. These compilers performed inlining at OPT with a fixed value of 100 for the threshold and 2000 for the limit.

As of z/OS V1R2 C/C++ compiler, the C++ compiler accepts the INLINE option, with defaults of 100 and 1000 for the threshold and limit, respectively. As a result

of this change, code that used to be inlined may no longer be inlined due to the decrease in the limit from 2000 to 1000 ACUs (Abstract Code Units).

As of z/OS V1R11 XL C/C++ compiler, the `INLINE` option might behave differently from those in the prior releases because of the implementation of a new inliner. You might find different performances of the `INLINE` option in the following ways:

- The functions that get inlined might be different.
- The inline report might look different.

If your application runs slower because functions that get inlined are different, adjust your inlining settings at high optimization levels, for example, the inlining threshold and the `#pragma inline/noinline` directives.

As of z/OS V2R1 XL C/C++ compiler, a virtual function might not be inlined even when the function is specified with the `always_inline` attribute. No informational message is issued when a virtual function is not inlined.

## IPA(LINK) compiler option

For detailed information about using IPA Link step, refer to IPA(LINK) in *z/OS XL C/C++ User's Guide*.

### IPA Link step default changes

As of OS/390 V1R3 C/C++ compiler, the following IPA Link step defaults changed:

- The default optimization level is `OPT(1)`.
- The default is `INLINE`, unless `NOOPT`, `OPT(0)` or `NOINLINE` is specified.

As of OS/390 V2R6 C/C++ compiler:

- The default optimization level for the IPA Link step is `OPT(2)`.
- The default inlining threshold is 1000 ACUs (Abstract Code Units). With OS/390 C/C++ V1R2 compiler, the threshold was 100 ACUs.
- The default expansion threshold is 8000 ACUs. With OS/390 C/C++ V1R2 C/C++ compiler, the threshold was 1000 ACUs.

### The IPA(LINK) option and exploitation of 64-bit virtual memory

As of z/OS V1R12 XL C/C++, the compiler component that executes IPA at both compile and link time is a 64-bit application, which will cause an XL C/C++ compiler ABEND if there is insufficient storage. The default `MEMLIMIT` system parameter size in the `SMFPRMxparmlib` member should be at least 3000 MB for the link, and 512 MB for the compile. The default `MEMLIMIT` value takes effect whenever the job does not specify one of the following:

- `MEMLIMIT` in the `JCL JOB` or `EXEC` statement
- `REGION=0` in the `JCL`

#### Notes:

- The compiler component that executes IPA(LINK) has been a 64-bit application since z/OS V1R8 XL C/C++ compiler.
- The `MEMLIMIT` value specified in an `IEFUSI` exit routine overrides all other `MEMLIMIT` settings.

The UNIX System Services `ulimit` command that is provided with z/OS can be used to set the `MEMLIMIT` default. For information, see *z/OS UNIX System Services*

*Command Reference.* For additional information about the MEMLIMIT system parameter, see *z/OS MVS Programming: Extended Addressability Guide*.

As of z/OS V1R8 XL C/C++ compiler, the EDCI, EDCXI, EDCQI, CBCI, CBCXI, and CBCQI cataloged procedures, which are used for IPA Link, contain the variable IMEMLIM, which can be used to override the default MEMLIMIT value.

### **IPA object module binary compatibility**

Release-to-release binary compatibility is maintained by the z/OS XL C/C++ IPA compilation and IPA link phases, as follows:

- An object file produced by an IPA compilation which contains IPA object or combined IPA and conventional object information can be used as input to the IPA link phase of the same or later version/release of the compiler.
- An object file produced by an IPA compilation which contains IPA object or combined IPA and conventional object information cannot be used as input by the IPA link phase of an earlier Version/Release of the compiler. If this is attempted, an error message will be issued by the IPA Link.
- If the IPA object is reproduced by a later IPA compilation, additional optimizations may be performed and the resulting application program might perform better.

**Exception:** The IPA object files produced by the OS/390 V1R2 C IPA compilation must be recompiled from the program source using an OS/390 V1R3 or later C/C++ compiler before you attempt to process them with the z/OS V2R1 XL C/C++ IPA Link.

## **LANGLVL(ANSI), LANGLVL(SAA), or LANGLVL(SAAL2) compiler option and macro redefinitions**

As of z/OS V1R7 XL C, the treatment of macro redefinitions has changed. For LANGLVL(ANSI), LANGLVL(SAA), or LANGLVL(SAAL2), the XL C compiler will issue a severe message instead of a warning message when a macro is redefined to a value that is different from the first definition.

```
#define COUNT 1
#define COUNT 2 /* error */
```

*Figure 9. Macro redefinition*

**Note:** Compare the treatment of macro redefinitions for these LANGLVL sub-options with that for “LANGLVL(EXTENDED) compiler option and macro redefinitions.”

## **LANGLVL(EXTENDED) compiler option and macro redefinitions**

As of z/OS V1R7 XL C, you can redefine a macro that has not been first undefined with LANGLVL(EXTENDED).

```
#define COUNT 1
#define COUNT 2

int main () {
    return COUNT;
}
```

*Figure 10. Macro redefinition under LANGLVL(EXTENDED)*

With z/OS V1R6 C and previous C compilers, this test will return "1". As of z/OS V1R7 XL C, this test will return "2".

**Note:** Compare the treatment of macro redefinitions for LANGLVL(EXTENDED) with that for "LANGLVL(ANSI), LANGLVL(SAA), or LANGLVL(SAAL2) compiler option and macro redefinitions" on page 66LANGLVL(ANSI), LANGLVL(SAA), or LANGLVL(SAAL2).

## LANGLVL(LONGLONG) compiler option

The long long data type is supported as a native data type when the LANGLVL(LONGLONG) option is turned on. This option is turned on by default by the compiler option LANGLVL(EXTENDED). The `_LONG_LONG` macro is predefined for all language levels other than ANSI.

As of z/OS V1R6 C/C++ compiler, when LANGLVL(LONGLONG) is turned on, the `_LONG_LONG` macro is defined by the compiler.

**Attention:** If you have defined your own `_LONG_LONG` macro in previous compiler releases, you must remove this user-defined macro before you compile your program.

## LOCALE compiler option

As of z/OS V1R9 XL C/C++, the `__LOCALE__` macro is defined to the name of the compile-time locale. If you specified `LOCALE(string literal)`, the compiler uses the runtime function `setlocale(LC_ALL "string literal")` to determine the name of the compile-time locale. If you do not use the `LOCALE` compiler option, the macro is undefined.

Prior to z/OS V1R9 XL C/C++, the `__LOCALE__` macro was defined to "" when the `LOCALE` option was specified without a suboption.

## M compiler option

Before z/OS V1R11, the stand-alone `makedepend` utility was used to analyze source files and determine source dependencies. As of z/OS V1R11, the `M` (`-qmakedep`) compiler option is introduced, and this compiler option is recommended to be used to obtain similar information. Specifying the `M` compiler option is equivalent to specifying the `-qmakedep` option.

The `M` compiler option is used to generate a make description file as a side-effect of the compilation process. The description file contains a rule or rules suitable for make that describes the dependencies of the main compilation source file. The `MF` option is used in conjunction with the `M` option and specifies the name of the file where the dependency information is generated, or the location of the file, or both. The `MF` option has no effect unless make dependency information is generated.

On z/OS systems, the `M` compiler option resolves a number of complexities that is not properly managed by the compiler-independent `makedepend` utility, thereby improving the accuracy of the dependency information.

For detailed information, refer to `MAKEDEP` compiler option in *z/OS XL C/C++ User's Guide*.

## OPTIMIZE compiler option

In the OS/390 V1R2, V1R3, V2R4, and V2R5 C/C++ compilers:

- OPT(0) mapped to NOOPT
- OPT and OPT(1) mapped to OPT(1)
- OPT(2) mapped to OPT(2)

As of OS/390 V2R6 C/C++:

- OPT(0) maps to NOOPT
- OPT, OPT(1) and OPT(2) map to OPT(2)

As of z/OS V1R5 C/C++, OPT(3) provides the compiler's highest and most aggressive level of optimization. OPT(3) is recommended only when the desire for runtime improvement outweighs the concern for minimizing compilation resources.

## NORENT compiler option

In previous releases of the compiler, **#pragma variable (name, RENT)** had no effect if the compiler option was NORENT. As of OS/390 V2R9 compiler, a variable can be reentrant even if the compiler option is NORENT. For more information, see “Reentrant variables when the compiler option is NORENT” on page 71.

## ROSTRING compiler option

As of z/OS V1R2 C/C++ compiler, the ROSTRING option default for C is changed from NOROSTRING to ROSTRING. The default for C++ has always been ROSTRING.

ROSTRING informs the compiler that string literals are read-only, thus allowing more freedom for the compiler to handle string literals. If you are not sure whether your program modifies string literals or not, specify the NOROSTRING compiler option.

## ROCONST compiler option

As of z/OS V1R2 C/C++ compiler, the ROCONST option default for C++ is changed from NOROCONST to ROCONST. The default for C remains NOROCONST.

As of OS/390 V2R10 C/C++ compiler, **#pragma variable (name, NORENT)** is accepted if the ROCONST option is turned on, and the variable is const-qualified and not initialized with an address. In previous releases, **#pragma variable (name, NORENT)** was ignored for static variables.

## STATICINLINE compiler option

As of z/OS V1R2 C/C++ compiler, the compiler supports the STATICINLINE compiler option. The default is NOSTATICINLINE. Specify STATICINLINE for compatibility with C++ compilers provided by previous versions of the compiler. For detailed information, refer to STATICINLINE compiler option in *z/OS XL C/C++ User's Guide*.

## SQL compiler option and SQL EXEC statements

See Chapter 21, “Migration issues with earlier C/C++ applications that use DB2 Universal Database,” on page 139.

## TARGET compiler option

As of z/OS V2R1 XL C/C++, the earliest release that can be targeted is z/OS V1R12 XL C/C++. For more information about the TARGET compiler option, refer to *z/OS XL C/C++ User's Guide*.

See also "Program modules from an earlier release" on page 101.

## TEST compiler option

As of z/OS V1R6 C/C++, when using the c89/c++ utility, the **-g** flag has changed from specifying the TEST option to DEBUG(FORMAT(DWARF)). For more information, see "Debug format specification" on page 96.

**Note:** Under ILP32 only, you can use the environment variable `{_DEBUG_FORMAT}` to determine the debug format (DWARF or ISD) to which the **-g** flag option is translated. For information about this environment variable and the c89/c++ utility, refer to the c89 utility information in *z/OS XL C/C++ User's Guide, SC14-7307*.

## TUNE compiler option

As of z/OS V2R1 XL C/C++ compiler, the default value of the TUNE compiler option is 7.

As of z/OS V1R6 C/C++ compiler, the default value of the TUNE compiler option is 5.

---

## Changes in IBM data set names

The names of IBM-supplied data sets may change from one release to another. See *z/OS Program Directory* for more information on data set names.

---

## Introduction of 1998 Standard C++ support

As of z/OS V1R2, the C++ compiler supports *Programming languages - C++ (ISO/IEC 14882:1998(E))*. See Part 5, "ISO Standard C++ compliance migration issues," on page 115 for details.

---

## Changes that affect performance and optimization

When you recompile OS/390 C/C++ programs with z/OS V2R1 XL C/C++ compiler, be aware of changes that you can make to improve performance.

## Addition of the #pragma reachable and #pragma leaves directives

The **#pragma reachable** and **#pragma leaves** directives help the optimizer in moving code around the function call site when exploring opportunities for optimization. Since the addition of these pragmas in OS/390 V2R9, the optimizer is more aggressive.

For more information on using #pragma reachable and #pragma leaves directives, refer to *z/OS XL C/C++ Language Reference, SC14-7308*.

---

## Changes that affect customized JCL procedures

The following topics apply if the JCL procedures that you are using either have been customized or should be customized.

### Potential increase in memory requirements

Memory requirements for compilation may increase for successive releases as new logic is added. If you cannot recompile an application that you successfully compiled with a previous release of the compiler, try increasing the region size. For the current default region size, refer to the *z/OS XL C/C++ User's Guide*.

As of z/OS V1R12 XL C/C++, when using the IPA compiler option to compile very large applications, you might need to increase the size of the work file associated with SYSUTIP DD in the IPA Link step. If you are linking the application in a USS environment, you can control the size of this work file with the `_CCN_IPA_WORK_SPACE` environment variable. If particularly large source files are compiled with IPA, the default size of the compile-time work files might also need to be increased. These can be modified via the `prefix_WORK_SPACE` environment variables.

### JCL CBCI and CBCXI procedures and the variable CLBPRFX

As of z/OS V1R5 C++ compiler, the CBCI and CBCXI procedures contain the variable CLBPRFX. If you have any JCL that uses these procedures, either they must be customized (for example, at installation time) or you must modify your JCL to provide a value for CLBPRFX.

### Syntax to invoke the CC command

With the C/C++ for MVS/ESA V3R2, OS/390, and z/OS XL C/C++ compilers, you can use a new syntax to invoke the CC command.

At customization time, your system programmer can customize the CC EXEC to accept only the old syntax (the one supported by compilers before C/C++ for MVS/ESA V3R2) compiler, only the new syntax, or both syntaxes.

The CC EXEC should be customized to accept only the new syntax.

If you customize the CC EXEC to accept both the old and new syntaxes, you must invoke it using either the old or the new syntax, not a mixture of both. Be aware that the old syntax does not support UNIX System Services files provided with z/OS.

Refer to the *z/OS Program Directory* for more information about installation and customization, and to the *z/OS XL C/C++ User's Guide* for more information about compiler options.

---

## Removal of Model Tool support

As of OS/390 V2R10 C/C++ compiler, Model Tool is no longer available.



---

## Chapter 11. Bind-time migration issues with OS/390 C/C++ programs

This information helps application programmers understand and resolve the compatibility issues that might occur when they relink programs from an OS/390 C/C++ compiler to z/OS V2R1 XL C/C++.

Executable program compatibility problems that require source changes are discussed in Chapter 9, "Source code compatibility issues with OS/390 programs," on page 59.

### Notes:

1. An executable program is the output of the prelink/link or bind process. For more information, see "Prelinking and linking of z/OS XL C/C++ programs" in *z/OS XL C/C++ User's Guide*.
2. The terms in this topic having to do with linking (bind, binding, link, link-edit) refer to the process of creating an executable program from object modules.
3. The output of a prelinking, linking, or binding process depends on where the programs are stored:
  - When the programs are stored in a PDS, the output is a *load module*.
  - When the programs are stored in a PDSE or in UNIX System Services files, the output is a *program object*.

When you bind programs that were previously compiled with an OS/390 compiler and library, be aware of the following potential migration issues:

- "Reentrant variables when the compiler option is NORENT"

---

### Reentrant variables when the compiler option is NORENT

If your program includes multithreaded operations, be aware of changes in the behavior of pragma variables.

In previous releases of the compiler, **#pragma variable (name, RENT)** had no effect if the compiler option was NORENT. As of OS/390 V2R9, a variable can be reentrant even if the compiler option is NORENT.

This change may cause some programs that compiled and linked successfully in previous releases to fail during link-edit in the current release. This applies if *all* of the following are true:

- The program is written in C and compiled with the NORENT option
- At least one variable is reentrant
- The program is compiled and linked with the output directed to a PDS and the prelinker was NOT used.

**Note:** JCL procedures that may have been used to do this in previous releases are: EDCCL, EDCCLG, EDCL, and EDCLG (not all of these procedures are available, starting with the z/OS V1R7 XL C/C++ compiler).



---

## Chapter 12. Runtime migration issues with OS/390 C/C++ applications

This information helps application programmers understand and resolve the compatibility issues that might occur when they relink programs from an OS/390 C/C++ compiler to z/OS V2R1 XL C/C++.

When you run applications that were previously compiled with an OS/390 compiler and library, be aware of the following potential migration issues:

- “Retention of OS/390 runtime behavior”
- “Debug format and translation of the c89 -g flag option” on page 74
- “Language Environment customization issues” on page 74

---

### Retention of OS/390 runtime behavior

When your program is using Language Environment services, you can use the ENVAR runtime option to specify the values of environment variables at execution time. You can use some environment variables to specify the original runtime behavior for particular items. The following setting specifies the original runtime behavior for the greatest number of items:

```
ENVAR("_EDC_COMPAT=32767")
```

Alternatively, you can add a call to the `setenv()` function, either in the CEEBINT High-Level Language exit routine or in your `main()` program. If you use CEEBINT only, you will need to relink your application. If you add a call to `setenv()` in the `main()` function, you must recompile the program and then relink your application. For more information, refer to `setenv()` in *z/OS XL C/C++ Runtime Library Reference, SC14-7314* and to Using environment variables in *z/OS XL C/C++ Programming Guide*.

### Changes to the `putenv()` function and POSIX compliance

As of z/OS V1R5 C/C++, the function `putenv()` places the string passed to `putenv()` directly into the array of environment variables. This behavior assures compliance with the POSIX standard.

Prior to z/OS V1R5 C/C++, the string used to define the environment variable passed into `putenv()` was not added to the array of environment variables. Instead, the system copied the string into system-allocated storage.

To allow the POSIX-compliant behavior of `putenv()`, do nothing; it's now the default condition.

To restore the previous behavior of `putenv()`, follow these steps:

1. Ensure that the environment variable, `_EDC_PUTENV_COPY`, is available on your pre-z/OS V1R5 system.
2. Set the environment variable `_EDC_PUTENV_COPY` to "YES".

For additional information, see:

- *z/OS XL C/C++ Runtime Library Reference*
- `_EDC_PUTENV_COPY` in *z/OS XL C/C++ Programming Guide*

---

## Debug format and translation of the c89 -g flag option

As of z/OS V1R6 C/C++, the environment variable `_DEBUG_FORMAT` can be used with the c89 utility to specify translation of the `-g` flag option for 31-bit compilations:

- If `_DEBUG_FORMAT` equals DWARF (the default), `-g` is translated to `DEBUG(FORMAT(DWARF))`.
- If `_DEBUG_FORMAT` equals ISD, then `-g` is translated to TEST (the old translation).

For the impact on specification of compiler options, see “Debug format specification” on page 96.

For more information about the c89 utility, see the c89 utility information in *z/OS XL C/C++ User's Guide*.

---

## Language Environment customization issues

For detailed information about customizing Language Environment runtime options, libraries, or processes, refer to *z/OS Language Environment Customization*.

---

## Change in allocation of VSAM control blocks

As of z/OS V1R10, the XL C/C++ compiler instructs VSAM, by default, to allocate control blocks and I/O buffers above the 16-MB line.

If you determine that this change could be causing a problem, you can use the VSAM JCL parameter AMP to override the default.

---

## Chapter 13. Migration issues resulting from class library changes between OS/390 C/C++ applications and Standard C++ library

Class library changes that have taken place since OS/390 C/C++ applications were developed have resulted in the following migration issues:

- “Function calls to different libraries”
- “Removal of IBM Open Class Library support”
- “Removal of Database Access Class Library utility”
- “Migration of programs with calls to UNIX System Laboratories I/O Stream Library functions”

---

### Function calls to different libraries

See “Function calls to different libraries” on page 79.

---

### Removal of IBM Open Class Library support

See “References to class libraries that are no longer shipped” on page 79.

---

### Removal of SOM support

As of OS/390 V2R10 C++ compiler, the IBM System Object Model (SOM) is no longer supported in the C++ compiler.

---

### Removal of Database Access Class Library utility

As of OS/390 V2R4 C++ compiler, the Database Access Class Library utility is no longer available.

---

### Migration of programs with calls to UNIX System Laboratories I/O Stream Library functions

See “Migration from UNIX System Laboratories I/O Stream Library to Standard C++ I/O Stream Library” on page 80.



---

## Part 4. Migration of earlier z/OS C/C++ applications to z/OS V2R1 XL C/C++

Earlier z/OS C/C++ applications were created with one of the following compilers:

- IBM z/OS V1R1 C/C++ (equivalent to the OS/390 V2R10 compiler)
- IBM z/OS V1R2 C/C++
- IBM z/OS V1R3 C/C++
- IBM z/OS V1R4 C/C++
- IBM z/OS V1R5 C/C++
- IBM z/OS V1R6 C/C++
- IBM z/OS V1R7 XL C/C++
- IBM z/OS V1R8 XL C/C++
- IBM z/OS V1R9 XL C/C++
- IBM z/OS V1R10 XL C/C++
- IBM z/OS V1R11 XL C/C++
- IBM z/OS V1R12 XL C/C++
- IBM z/OS V1R13 XL C/C++

**Note:** The z/OS V1R3 and V1R4 compilers are equivalent to the z/OS V1R2 compiler.

Significant class library changes occurred with releases z/OS V1R5 C/C++ through z/OS V1R9 XL C/C++. These changes could necessitate changes in your source code.

### Notes:

1. If your application uses IBM CICS information or statements, also see Chapter 20, “Migration issues with earlier C/C++ applications that run CICS statements,” on page 133.
2. If your application uses IBM DB2 information or statements, also see Chapter 21, “Migration issues with earlier C/C++ applications that use DB2 Universal Database,” on page 139.

The following topics provide information relevant to migrating an earlier z/OS C/C++ application to z/OS V2R1 XL C/C++:

- Chapter 14, “Source code compatibility issues with earlier z/OS C/C++ programs,” on page 79
- Chapter 15, “Compile-time migration issues with earlier z/OS C/C++ programs,” on page 87
- Chapter 16, “Bind-time migration issues with earlier z/OS C/C++ programs,” on page 101
- Chapter 17, “Runtime migration issues with earlier z/OS C/C++ applications,” on page 105





---

## Chapter 14. Source code compatibility issues with earlier z/OS C/C++ programs

Significant class library changes have occurred between z/OS V1R5 C/C++ compiler and z/OS V2R1 XL C/C++ compiler. These changes could necessitate changes in your source code. Otherwise, you can likely use source programs that were created with one of the earlier z/OS C/C++ compilers without modification.

Exceptions are highlighted in the following topics:

- “Function calls to different libraries”
- “References to class libraries that are no longer shipped”
- “Migration from UNIX System Laboratories I/O Stream Library to Standard C++ I/O Stream Library” on page 80
- “Standard C++ compliance compatibility issues” on page 80
- “Use of XL C/C++ library functions” on page 80
- “Use of pragmas” on page 83
- “Virtual function declaration and use” on page 84

**Note:** Some source code compatibility issues can be addressed by modifying runtime options. See Chapter 12, “Runtime migration issues with OS/390 C/C++ applications,” on page 73.

---

### Function calls to different libraries

While it is possible to use functions from more than one library, (Standard C++ I/O Stream Library, UNIX System Laboratories I/O Stream Library, and C I/O), it is not recommended because it requires that your code perform extra tasks. For example, the UNIX System Laboratories I/O Stream Library uses a separate buffer so you would need to flush the buffer after each call to `cout` by either setting `ios::unitbuf` or calling `sync_with_stdio()`.

You should avoid switching between the I/O Stream Library formatted extraction functions and C `stdio.h` library functions whenever possible, and you should also avoid switching between versions of the I/O Stream Libraries. For more information, see *z/OS XL C/C++ Programming Guide, SC14-7315*.

---

### References to class libraries that are no longer shipped

As of z/OS V1R9, IBM Open Class Library (IOC) dynamic link libraries (DLLs) are no longer shipped with the z/OS XL C/C++ compiler.

Any source dependency on an IOC DLL must be removed.

For information about the libraries that are supported by the current release, see *z/OS XL C/C++ Runtime Library Reference*.

---

## Migration from UNIX System Laboratories I/O Stream Library to Standard C++ I/O Stream Library

The values for some enumerations differ slightly between the UNIX System Laboratories and Standard C++ I/O Stream Library. This may cause problems when migrating programs to the Standard C++ I/O Stream Library.

The following IOS format flags have been added to the Standard C++ I/O Stream Library:

- `boolalpha`
- `adjustfield`
- `basefield`
- `floatfield`

The following IOS format flags have been removed:

- flags for format control: `stdio`
- flags for open-mode control: `nocreate`, `noreplace`, `bin`
- flags for the io-state control: `hardfail`

There might be other small differences.

---

## Standard C++ compliance compatibility issues

As of z/OS V1R7, the XL C++ compiler supports *Programming languages - C++ (ISO/IEC 14882:2003(E))*, which documents the currently supported Standard C++. For more information, see Part 5, “ISO Standard C++ compliance migration issues,” on page 115.

---

## Use of XL C/C++ library functions

The use of XL C/C++ library functions can be affected by performance enhancements such as:

- “Timing of processor release by the `pthread_yield()` function”
- “New information returned by the `getnameinfo()` function” on page 81

as well as by changes to external standards, such as:

- “Feature test macros and system header files” on page 81
- “Potential need to include `_Ieee754.h`” on page 81
- “New definitions exposed by use of the `_OPEN_SYS_SOCKET_IPV6` macro” on page 82
- “Required changes to `fprintf` and `fscanf` strings `%D`, `%DD`, and `%H`” on page 82
- “Changes to the `putenv()` function and POSIX compliance” on page 82

### Timing of processor release by the `pthread_yield()` function

As of z/OS V1R8 XL C/C++ compiler, the `_EDC_PTHREAD_YIELD` environment variable can be used to either release the processor immediately, or release the processor after a delay. This change affects both the `pthread_yield()` and `sched_yield()` functions.

In prior releases, control was passed back to the calling thread without releasing the processor whenever multiple intra-thread calls to `pthread_yield()` occurred within .01 seconds of one another.

If you want to continue to use the previous internal timing algorithm, use the following command:

```
_EDC_PTHREAD_YIELD=-1
```

For information about `_EDC_PTHREAD_YIELD` and setting environment variables, see Using environment variables in *z/OS XL C/C++ Programming Guide, SC14-7315*.

For information about the `pthread_yield()` and `sched_yield()` functions, see *z/OS XL C/C++ Runtime Library Reference, SC14-7314*.

## New information returned by the `getnameinfo()` function

As of z/OS V1R9 XL C/C++ compiler, invocations of the `getnameinfo()` function might need to be modified to handle interface information appended to the host name. Prior to z/OS V1R9, the `getnameinfo()` function ignored the zone index value in the input `sockaddr_in6` structure.

Ensure that you verify the capability to handle scope information of `getnameinfo()` invocations that have the following characteristics:

- The `sa` argument represents an IPv6 link-local address.
- The `sin6_scope_id` member of `sa` is non-zero.

The scope information is returned in the format *hostname%interface*. The host name is the node name associated with the IP address in the buffer pointed to by the host argument. By default, the scope information is the interface name associated with the zone index value.

For information about options for addressing this change, see Communications Server migration actions in *z/OS Migration, GA32-0889*.

For information about the `getnameinfo()` function, see *z/OS XL C/C++ Runtime Library Reference, SC14-7314*.

## Feature test macros and system header files

You must define the feature test macros that you need before including any system headers.

Feature test macros control which symbols are made visible in a source file (typically a header file). For detailed information about header files and supported feature test macros, see *z/OS XL C/C++ Runtime Library Reference, SC14-7314*.

## Potential need to include `_IEEE754.h`

As of z/OS XL C/C++ V1R9 compiler, the `<math.h>` file (included in the `<tgmath.h>` header file) no longer includes the `<_IEEE754.h>` file, which declares IEEE 754 interfaces.

This change avoids potential namespace pollution. If your code needs any symbols that are defined in `<_IEEE754.h>`, you must explicitly include that header file.

For additional information about runtime library support of decimal floating-point data types and functions, see *z/OS XL C/C++ Runtime Library Reference, SC14-7314*.

## New definitions exposed by use of the `_OPEN_SYS_SOCKET_IPV6` macro

As of z/OS V1R7 XL C/C++ compiler, recompiling an earlier C/C++ program that uses the `_OPEN_SYS_SOCKET_IPV6` feature test macro will expose new definitions in the system header files as well as new functions in `netinet/in.h`. These new functions are:

```
inet6_opt_append() inet6_opt_find() inet6_opt_finish() inet6_opt_get_val()
inet6_opt_init()   inet6_opt_next() inet6_opt_set_val() inet6_rth_add()
inet6_rth_getaddr() inet6_rth_init() inet6_rth_reverse() inet6_rth_segments()
inet6_rth_space()
```

## Required changes to `fprintf` and `fscanf` strings `%D`, `%DD`, and `%H`

As of z/OS V1R8, XL C/C++ supports decimal floating point size modifiers ("`D`", "`DD`", and "`H`") for the `fprintf` and `fscanf` families of functions. If a percent sign (`%`) is followed by one of these character strings, which had no meaning under previous releases of z/OS XL C/C++, the compiler could interpret the data as a size modifier. Treatment of this condition is undefined and the behavior could be unexpected.

For a description of the potential results, see "Unexpected output from `fprintf()` or `fscanf()`" on page 106.

If you are using z/OS V1R9 XL C/C++ compiler and you want the `fprintf()` and `fscanf()` families of functions to produce the same results as your previous compiler, change your source code input as shown in Table 13.

Table 13. Example: Code change for `fprintf/fscanf` character strings "`%D`", "`%DD`", and "`%H`"

Existing statement	Modification required under z/OS V2R1 XL C/C++
<code>printf("This results in a 10%Deduction.\n");</code>	<code>printf("This results in a 10%DDeduction.\n");</code>

## Changes to the `putenv()` function and POSIX compliance

As of z/OS V1R5 C/C++, the function `putenv()` places the string passed to `putenv()` directly into the array of environment variables. This behavior assures compliance with the POSIX standard.

Prior to z/OS V1R5 C/C++, the string used to define the environment variable passed into `putenv()` was not added to the array of environment variables. Instead, the system copied the string into system-allocated storage.

To allow the POSIX-compliant behavior of `putenv()`, do nothing; it's now the default condition.

To restore the previous behavior of `putenv()`, follow these steps:

1. Ensure that the environment variable, `_EDC_PUTENV_COPY`, is available on your pre-z/OS V1R5 system.
2. Set the environment variable `_EDC_PUTENV_COPY` to "YES".

For additional information, see:

- *z/OS XL C/C++ Runtime Library Reference*

- `_EDC_PUTENV_COPY` in *z/OS XL C/C++ Programming Guide*

---

## C99 support of long long data type

As of z/OS V1R7 XL C/C++ compiler, when you recompile an application that uses long long support, you might experience problems if the application does one of the following actions:

- Uses a compiler designed to support C99
- Does not ask for extended features

If an application currently uses the `LANGLVL(LONGLONG)` compiler option to get at the long long data type, and also uses certain non-standard long long macros, recompiling with z/OS V2R1 XL C/C++ may cause compiler error messages to be issued because these non-standard definitions are hidden unless both `LANGLVL(LONGLONG)` and `LANGLVL(EXTENDED)` are in effect.

If an application currently uses `LANGLVL(EXTENDED)`, the non-standard definitions will continue to be exposed since extended features are requested. For those applications that want to use a compiler designed to support C99, but do not want extended features, change the source code to use the C99 standard long long macros, as shown in Table 14.

*Table 14. C99 standard macros to replace non-standard long long macros that cause z/OS V2R1 errors*

Non-standard long long macros	C99 standard long long macros
<code>LONGLONG_MIN</code>	<code>LLONG_MIN</code>
<code>LONGLONG_MAX</code>	<code>LLONG_MAX</code>
<code>ULONGLONG_MAX</code>	<code>ULLONG_MAX</code>

The definitions in Table 14 are commonly used with the following functions:

- `llabs()`
- the following long long numeric conversion functions
  - `strtoll()`
  - `strtoull()`
  - `wcstoll()`
  - `wcstoull()`

---

## Use of pragmas

Functionality of pragmas can change from release to release, or under different circumstances. Be aware of the following migration issues:

- “Application of `#pragma unroll()` as of z/OS V1R7 XL C/C++”
- “Unexpected C++ output with `#pragma pack(2)`” on page 84

### Application of `#pragma unroll()` as of z/OS V1R7 XL C/C++

As of z/OS V1R7 XL C/C++ compiler, the `#pragma unroll()` directive works only with for loops.

If your code applies the `#pragma unroll()` directive to a while or a do loop, the compiler ignores the pragma directive and generates a warning message.

For detailed information about unrolling loops, refer to any or all of the following related documents:

- *z/OS XL C/C++ Language Reference, SC14-7308*
- *z/OS XL C/C++ Programming Guide, SC14-7315*
- *z/OS XL C/C++ User's Guide, SC14-7307*

## Unexpected C++ output with #pragma pack(2)

An aggregate, which contains char data type members only, has natural alignment of one byte. XL C retains the natural one-byte alignment but when **#pragma pack(2)** is applied to an aggregate, its alignment increases to two bytes.

If XL C and XL C++ program objects need to be compatible, do not use **#pragma pack(2)** in your XL C or XL C++ code.

**Note:** You can use the sizeof operator to test the output whenever **#pragma pack(2)** is used.

For more information about **#pragma pack(2)**, refer to the discussion of the **#pragma pack** directive **twobyte** option in *z/OS XL C/C++ Language Reference*.

---

## Virtual function declaration and use

Figure 11 on page 85 shows a program that, as of z/OS V1R6 C/C++ compiler, would generate an exception under the IBM object model because the call to a member function `version()` on the object `_b` occurs before the declaration of `_b`.

```

#include

class A {
    public:
        A(int i) : v(i) {}
        virtual int version() {return 0;} 1;
        private: int v;
};

class B:virtual public A {
    public:
        B(int i) : A(i) {}
};

extern B _b; 2
static int ver = _b.version(); 3
B _b(1); 4

int main() {
    printf("version: %d\n", ver);
    return 0;
}

```

**Notes:**

1. The virtual keyword tells the compiler that the function is virtual and it can be overloaded by any derived class of A.
2. A reference to externally defined `_b` of type B.
3. The value of static global variable `ver` is initialized with the value returned by member function `version()` called by object `_b`. An exception will be raised because the object `_b` is not fully constructed at the time of the call to the member function `version()`.
4. The declaration of the polymorphic object `_b` occurs after its use on the previous line. This line should precede the definition of `ver` to ensure that the virtual function `version()` is found at run time.

*Figure 11. Example that highlights sequence of statements to declare and call a virtual function*





---

## Chapter 15. Compile-time migration issues with earlier z/OS C/C++ programs

When you compile earlier z/OS C/C++ programs with z/OS V2R1 XL C/C++, be aware of the following information:

- “Changes in compiler listings, messages, and return codes”
- “Changes in compiler option functionality” on page 91
- “Changes that affect compiler invocations” on page 95
- “Changes that affect JCL procedures” on page 97
- “JCL that runs pre-z/OS V1R5 C/C++ programs” on page 99
- “Compiler options that manage Standard C++ compliance” on page 99
- “Impact of recompiling applications that include `<net/if.h>` with the `_XOPEN_SOURCE_EXTENDED` feature test macro” on page 100
- “Impact of recompiling applications that include the `pselect()` interface” on page 100
- “Impact of recompiling with the `_OPEN_SYS_SOCKET_IPV6` macro” on page 100
- “Impact of recompiling code that relies on `math.h` to include IEEE 754 interfaces” on page 100

---

### Changes in compiler listings, messages, and return codes

From release to release, message contents can change and, for some messages, return codes can change. Errors can become warnings, and warnings can become errors. You must update any application that is affected by changes in message contents or return codes. Do not build dependencies on message contents, message numbers, or return codes. See *z/OS XL C/C++ Messages* for a list of compiler messages.

Listing formats, especially the pseudo-assembler parts, will continue to change from release to release. Do not build dependencies on the structure or content of listings. For information about C listings or the C++ listings for the current release, refer to *z/OS XL C/C++ User's Guide, SC14-7307*.

You might need to be aware of changes with respect to the following issues:

- “Appearance of compiler substitution variables” on page 88
- “Function offsets in source listing” on page 88
- “Diagnostic refinement in identification of linkage issues (C++ only)” on page 88
- “References to UNIX System Services file names” on page 89
- “Non-compliant array index raises an exception” on page 89
- “Unexpected name lookup error messages with template use” on page 90
- “Width of mnemonic in assembly listings” on page 90
- “Macro redefinitions and error messages” on page 90

For information about the language levels that are affected, see “LANGLVL(ANSI), LANGLVL(SAA), or LANGLVL(SAAL2) compiler option and macro redefinitions” on page 93 and “LANGLVL(EXTENDED) compiler option and macro redefinitions” on page 94.

## Appearance of compiler substitution variables

As of z/OS V1R10, the compiler substitution variable appears, where applicable, in the message section of a compilation listing. This is to avoid the confusion that can be caused by a string of blank spaces in the listing.

## Corrections in escape sequence encoding

As of z/OS V1R11, the encoding of octal escape characters in string literals and wide string literals is corrected. See the corrected processing in the following table (where the bytecode is shown using base 16).

Table 15. Corrections in escape sequence encoding

Example	Old bytecode (INCORRECT)	New bytecode (CORRECT)	Description
"\776"	01fe00	fe00	Octal escape overflow in narrow string literals.
L"\776"	0001fe00 00	01fe0000	Octal escape above \377 (no overflow) in wide string literal.

## Function offsets in source listing

As of z/OS V1R10, the XL C/C++ compiler adds the starting offset of each function to the listing when the OFFSET option is specified.

## Diagnostic refinement in identification of linkage issues (C++ only)

Prior to z/OS V1R9 XL C/C++ PTF UK31348, the XL C++ compiler diagnosed any case in which two functions with the same linkage signature were mapped together. For examples, see Figure 12 and Figure 13 on page 89.

As of z/OS V1R9 XL C/C++ PTF UK31348, the XL C++ compiler diagnoses two functions that are mapped together only when both are defined in the same compilation unit, without considering differences in linkage signature. See Figure 14 on page 89.

```
// t.C
extern "C" int foo(int);
extern "C" int bar(double);
#pragma map (foo, "bar")
int f() { return foo(2) + bar(3.0);}
```

Figure 12. Example of diagnosis of two externally defined functions with different types mapped together, prior to z/OS V1R9 XL C/C++ PTF UK31348

The diagnostic message will identify the mapping of foo with "bar" as invalid because their declarations differ in type.

```
// t.C
int foo(double);
extern "C" int bar(double);
#pragma map (foo, "bar")
int f() { return foo(2) + bar(3.0);}
```

*Figure 13. Example of diagnosis of two externally defined functions with different linkage signatures mapped together, prior to z/OS V1R9 XL C/C++ PTF UK31348*

The diagnostic message will identify the mapping of foo with "bar" as invalid because, although they are defined with the same type, one is defined with a default linkage.

```
// t.C
extern "C" int foo(int) { return 0; }
extern "C" int bar(int) { return 2.0; }
#pragma map (foo, "bar")
int f() { return foo(2) + bar(3.0);}
```

*Figure 14. Example of diagnosis of two functions with the same linkage signatures mapped together as of z/OS V1R9 XL C/C++ with PTF UK31348 applied*

The diagnostic message will identify the mapping of foo with "bar" as invalid because both are defined, which violates the one-definition rule.

## References to UNIX System Services file names

As of z/OS V1R9, when compiling C source files that reside in the UNIX System Services file system, any messages emitted during the compilation will use relative path information, rather than absolute path information, to reference the file name. This makes all file-name references in the compiler error messages and listings consistent in that they all use relative path information.

## Non-compliant array index raises an exception

As of z/OS V1R9 XL C++, an error message is generated whenever an array index is defined as anything other than an integral non-volatile constant expression. This change alerts you that your code does not comply with the currently supported C++ Standard (section 5.19). For an example, see Figure 15.

### Notes:

1. To avoid this problem, redefine the array index to an integral non-volatile constant expression.
2. Prior to z/OS V1R9 XL C++, the compiler allowed local validation of this rule.

```
void f() {}
int main()
{
int i[(int)f];
return 0;
}
```

*Figure 15. Example of volatile array index*

The compiler will generate a message stating that the expression must be an integral non-volatile constant expression.

## Unexpected name lookup error messages with template use

As of z/OS V1R9 XL C++ compiler, new name lookup exceptions could result from compiling a template which uses symbolic names that do not depend on that template's parameters. For an example, see Figure 16 and Figure 17.

Symbolic names that are not dependent on a template parameter must be:

- Declared before they are used.
- Defined before they are used in a context that requires a complete definition.

Earlier releases allowed names to be used in a template definition before they were declared as long as they were declared before the template was instantiated.

**Note:** This change will not affect well-formed code, which always defines names in the source code before using them.

For information about using templates in C++ programs, see *z/OS XL C/C++ Programming Guide, SC14-7315*. For information about compiling, binding, and running C++ templates, see *z/OS XL C/C++ User's Guide*.

```
template <class T> void fnc(T &x, T y)
{
    int t1=FAIL;
    int t2=ZERO;
    int t3=ONE;
}

enum ENUMTYPE {ZERO = 3, ONE, FAIL} e1, e2, e3, e4;

struct tst{};

template void fnc(tst &x, tst y);
```

Figure 16. Example of C++ template code that will cause name lookup exceptions.

If the compiler encounters this code before it encounters the declarations of the symbolic names FAIL, ZERO, and ONE, it will generate the messages listed in Figure 17.

```
"/ex1.cpp", line 3.11: CCN5274 (S) The name lookup for "FAIL" did not find a declaration.
"/ex1.cpp", line 8.31: CCN6303 (I) "ENUMTYPE FAIL" is not visible.
"/ex1.cpp", line 1.25: CCN5700 (I) The previous message was produced while
processing "fnc(tst &, tst)".
"/ex1.cpp", line 4.11: CCN5274 (S) The name lookup for "ZERO" did not find a declaration.
"/ex1.cpp", line 8.16: CCN6303 (I) "ENUMTYPE ZERO" is not visible.
"/ex1.cpp", line 5.11: CCN5274 (S) The name lookup for "ONE" did not find a declaration.
"/ex1.cpp", line 8.26: CCN6303 (I) "ENUMTYPE ONE" is not visible.
```

Figure 17. Messages that result from attempts to compile the code in Figure 16.

## Width of mnemonic in assembly listings

As of z/OS V1R9 XL C/C++ compiler, customized JCL procedures or other tools that scan assembly listings might need to be updated because the width of the instruction mnemonic has been increased.

## Macro redefinitions and error messages

As of z/OS V1R7 XL C, the behavior of macro redefinition has changed. For certain language levels, the XL C compiler will issue a severe error message instead of a warning message when a macro is redefined to a value that is different from the first definition.

---

## Changes in compiler option functionality

The following topics describe changes in compiler option functionality that might require modifications to either your use of compiler options or your source code. For detailed information about these compiler options, see *z/OS XL C/C++ User's Guide, SC14-7307*.

### Option behavior change when processing multiple suboptions

As of z/OS V2R1, when multiple suboptions are specified with the following options, the compiler no longer issues a diagnostic message, and the last suboption is used:

- AGGRCOPY
- ASSERT
- CHECKOUT
- DLL
- PORT
- PPOONLY

### CHECKOUT compiler option

Starting from z/OS V1R13, the CHECKOUT option is deprecated. Use the INFO option instead of CHECKOUT.

### CMDOPTS compiler option and conflict resolution

As of z/OS V1R7 XL C/C++ compiler:

- Default options specified in the configuration file have the same weight as if they were specified on the command line. The XL C/C++ compiler cannot distinguish between an option specified in the configuration file and an option specified on the command line.
- Any conflict between options and pragmas is resolved in favor of the option.
- The XL C/C++ compiler no longer requires that default options be specified in the configuration file.

As of z/OS V1R7 XL C/C++, if you customize your xlc configuration file using the sample default configuration file, you might experience a change in behavior because the defaults for supported xlc commands are no longer specified in the options attribute in the configuration file. Instead, the xlc utility emits the defaults as suboptions of the CMDOPTS compiler option. This may cause a change in behavior because the XL C/C++ compiler resolves conflicts between options and pragmas differently, depending on whether options are specified as suboptions of the CMDOPTS option or explicitly on the command line and in the options attributes.

### DFP compiler option and earlier floating-point applications

As of z/OS V1R10, there is a risk that earlier C/C++ applications compiled with the DFP option could inadvertently reset the decimal floating-point rounding mode to the default value. You should consider this risk if you are adding decimal floating-point functionality to an application that includes floating-point operations which use the data type `fenv_t` or the function `fesetenv()` with the static initializer `FE_DFL_ENV`. This is because the `FE_DFL_ENV` and `__fe_def_env` static initializers set the decimal floating-point rounding mode to the `FE_DEC_TONEAREST` value.

Be aware of the following constraints

- Because the decimal floating-point rounding mode field is stored in the FPC register separately from the binary floating-point rounding mode, there will be no effect on the binary floating-point rounding mode. However, you should take care with exception handling routines because binary floating-point applications can use FPC exception flags.
- DFP names will not be exposed when the application is compiled without the DFP compiler option. (There may also be a new `__STDC_WANT_DEC_FP__` C99 feature test macro to further protect against namespace invasion).
- If you are compiling a System Programming C (SPC) application, you should not use the DFP option; the statically bound version of the SPC function `sprintf()` does not support decimal floating-point number formats. Standard functions that are already supported in the SPC library (such as `printf()` and `scanf()`) will be able to operate on decimal floating-point numbers.

## DSAUSER compiler option

Starting from z/OS V1R13 XL C++ compiler, the DSAUSER option is supported. When the METAL option is in effect, the DSAUSER option requests a user field of the size of a pointer to be reserved on the stack. The default is NODSAUSER. For more information, see **DSAUSER | NODSAUSER (C only)** that is documented in *z/OS XL C/C++ User's Guide*.

## ENUMSIZE(SMALL) and protected enumeration types in system header files

As of z/OS V1R7 XL C/C++ compiler, selected enumerated (enum) type declarations in system header files are protected to avoid potential execution errors. This allows you to specify the ENUMSIZE compiler option with a value other than SMALL without risking incorrect mapping of enum data types (for example, if they were used inside of a structure).

With earlier versions of the compiler, if you specified ENUMSIZE() with a value other than SMALL, data that was declared with certain enum types could be incorrectly mapped. In some instances, the header files in the library referenced the types (such as `__device_t` in the typedef `fldata_t`), which resulted in a potential inconsistency between the mapping seen during application execution and that declared in the library (which is built with the default ENUMSIZE(SMALL)).

Even when you specify ENUMSIZE with a value other than SMALL, the enumerations listed in Table 16 will always be ENUMSIZE(SMALL).

Table 16. Header files with declarations of protected enumeration types

Header file	Enumerations
stdio.h	<code>__device_t</code>
search.h	<code>ACTION</code> <code>VISIT</code>
sys/uiio.h	<code>uio_rw</code>
sys/wait.h	<code>idtype_t</code>
<code>_Ccsid.h</code>	<code>__csType</code>
<code>__ledebug.h</code>	<code>asfAmodeType</code> <code>asfCallbackResult</code>
yvals.h	<code>_Mux</code>

## FLAG compiler option

As of z/OS V1R13, FLAG(I) is the default in z/OS UNIX System Services as it is in batch compilation.

## FLOAT(AFP) suboptions for applications that access CICS data

See “CICS TS V4.1 with "Extended MVS Linkage Convention"” on page 136.

## GENASM compiler option

Starting from z/OS V1R13, the GENASM option is not supported in UNIX System Services. Instead, you can use the -S flag and the -o option.

## GONUMBER compiler option and LP64 support

As of z/OS V1R8 XL C/C++ compiler, the GONUMBER compiler option generates line number tables for both 31-bit and 64-bit applications.

## IPA compiler option

Prior to z/OS V2R1, the default optimization level for the IPA option was NOOPTIMIZE when the compiler was invoked from JCL.

Starting with z/OS V2R1, the default optimization level for the IPA option is OPTIMIZE(2) when the compiler is invoked from JCL. This change was made to match the default optimization level when the compiler is invoked from USS, as well as the default on the other platforms.

## LANGLVL(ANSI), LANGLVL(SAA), or LANGLVL(SAAL2) compiler option and macro redefinitions

As of z/OS V1R7 XL C, the treatment of macro redefinitions has changed. For LANGLVL(ANSI), LANGLVL(SAA), or LANGLVL(SAAL2), the XL C compiler will issue a severe message instead of a warning message when a macro is redefined to a value that is different from the first definition.

```
#define COUNT 1
#define COUNT 2 /* error */
```

*Figure 18. Macro redefinition*

**Note:** Compare the treatment of macro redefinitions for these LANGLVL sub-options with that in “LANGLVL(EXTENDED) compiler option and macro redefinitions” on page 94.

## LANGLVL(EXTC1X) compiler option

This option controls that compilation is based on the C11 standard, invoking all the currently supported C11 features and other implementation-specific language extensions. For detailed information, see EXTC1X that is documented in *z/OS XL C/C++ User's Guide*.

**Note:** C11 is a new version of the C programming language standard. IBM continues to develop and implement the features of the new standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the features of the C11 standard is complete, including the support of a new C standard library, the implementation may change from release to release. IBM makes no attempt to maintain

compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new features of the C11 standard and therefore they should not be relied on as a stable programming interface.

## LANGLVL(EXTENDED) compiler option and macro redefinitions

As of z/OS V1R7 XL C, you can redefine a macro that has not been first undefined with LANTLRVL(EXTENDED).

```
#define COUNT 1
#define COUNT 2

int main () {
    return COUNT;
}
```

Figure 19. Macro redefinition under LANTLRVL(EXTENDED)

With z/OS V1R6 C and previous C compilers, this test will return "1". As of z/OS V1R7 XL C, this test will return "2".

**Note:** Compare the treatment of macro redefinitions for LANTLRVL(EXTENDED) with that for "LANTLRVL(ANSI), LANTLRVL(SAA), or LANTLRVL(SAAL2) compiler option and macro redefinitions" on page 93.

## LANGLVL(EXTENDED0X) compiler option

This option controls that compilation is based on the C++11 standard, invoking all the currently supported C++11 features and other implementation-specific language extensions. The option is implemented in XL C/C++ compiler as of z/OS V1R11. For detailed information, see **LANGLVL(EXTENDED0X) compiler option** that is documented in *z/OS XL C/C++ User's Guide*.

**Note:** C++11 is a new version of the C++ programming language standard. IBM continues to develop and implement the features of the new standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the features of the C++11 standard is complete, including the support of a new C++ standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new features of the C++11 standard and therefore they should not be relied on as a stable programming interface.

## LOCALE compiler option

As of z/OS V1R9 XL C/C++, the `__LOCALE__` macro is defined to the name of the compile-time locale. If you specified `LOCALE(string literal)`, the compiler uses the runtime function `setlocale(LC_ALL "string literal")` to determine the name of the compile-time locale. If you do not use the `LOCALE` compiler option, the macro is undefined.

Prior to z/OS V1R9 XL C/C++, the `__LOCALE__` macro was defined to "" when the `LOCALE` option was specified without a suboption.



## M compiler option

Before z/OS V1R11, the stand-alone makedepend utility was used to analyze source files and determine source dependencies. As of z/OS V1R11, the M (-qmakedep) compiler option is introduced to provide similar information.

The M compiler option is used to generate a "make" description file as a side-effect of the compilation process. The description file contains a rule or rules suitable for "make" that describes the dependencies of the main compilation source file. The MF suboption is used in conjunction with the M compiler option and specifies the name of the file where the dependency information is generated, or the location of the file, or both.

For detailed information, refer to MAKEDEP compiler option in *z/OS XL C/C++ User's Guide*.

## RESTRICT option

z/OS V1R12 XL C compiler introduces a new option RESTRICT to indicate to the compiler that all pointer parameters in some or all functions are disjoint. The default is NORESTRICT. For detailed information, see RESTRICT | NORESTRICT (C only) in *z/OS XL C/C++ User's Guide*.

## SEVERITY option

z/OS V1R12 XL C compiler introduces a new option SEVERITY to support message severity modification. With this option specified, you can set the severity level for a certain message that you specified. The compiler will use the new severity when the specified message is generated by the compiler. The default is NOSEVERITY. For detailed information, see SEVERITY | NOSEVERITY (C only) in *z/OS XL C/C++ User's Guide*.

## SQL compiler option and SQL EXEC statements

See Chapter 21, "Migration issues with earlier C/C++ applications that use DB2 Universal Database," on page 139.

## TARGET compiler option

As of z/OS V2R1 XL C/C++, the earliest release that can be targeted is z/OS V1R12 XL C/C++. For more information about the TARGET compiler option, refer to *z/OS XL C/C++ User's Guide*.

See also "Program modules from an earlier release" on page 101.

## TEMPLATEDEPTH compiler option

Starting from z/OS V1R13 XL C++ compiler, the TEMPLATEDEPTH option is supported. With this option, you can specify the maximum number of recursively instantiated template specializations that are processed by the compiler. The default is TEMPLATEDEPTH(300). For more information, see **TEMPLATEDEPTH (C++ only)** that is documented in *z/OS XL C/C++ User's Guide*.

---

## Changes that affect compiler invocations

As of z/OS V1R6 C/C++ compiler, compiler invocation is supported by two different utilities:

- c89
- xlc

z/OS V1R6 C/C++ introduced the following utilities:

- **x1c** command, to compile a C program
- **x1c** and **x1c++** commands, to compile a C++ program

z/OS V1R6 C/C++ introduced the following command suffixes:

- **\_x** suffix, which compiles the program with XPLINK
- **\_64** suffix, which compiles the program under LP64

The utility you want to use depends on:

- Whether you need to port code between z/OS and AIX®.
- How you want to set up your build environment.

For example, you can use the command **c89\_x** to compile an ANSI-compliant program with XPLINK.

**Note:** As of z/OS V1R7 XL C/C++, you no longer need to use command names with suffixes **\_x/\_64** to compile/bind an XPLINK or 64-bit application. You can use suffixless command names with **-qxplink/-q64** or **-Wc,xplink/-Wc,lp64** and **-Wl,xplink/-Wl,lp64** instead. For detailed information, refer to the c89 utility information in *z/OS XL C/C++ User's Guide*.

Table 17. Differences between the c89 and xlc compiler invocation utilities

	<b>c89 utility</b>	<b>xlc utility</b>
<b>Command support</b>	<p>The c89 utility does not support</p> <ul style="list-style-type: none"> <li>• The <b>-S</b> flag option introduced in z/OS V1R9.</li> <li>• AIX options syntax.</li> </ul>	<p>The following commands accept AIX C/C++ as well as z/OS C/C++ options syntax:</p> <ul style="list-style-type: none"> <li>• <b>cc</b></li> <li>• <b>c89</b></li> <li>• <b>cxx</b></li> <li>• <b>c++</b></li> </ul> <p>The xlc utility does not support the TEMPINC compiler option.</p>
<b>Environment setup</b>	Determined by environment variables	Determined by configuration file

## Changes that affect use of the c89 command

### Debug format specification

As of z/OS V1R6 C/C++, the environment variable **\_DEBUG\_FORMAT** can be used with the c89 utility to specify translation of the **-g** flag option for 31-bit compilations:

- If **\_DEBUG\_FORMAT** equals DWARF (the default), **-g** is translated to **DEBUG(FORMAT(DWARF))**.
- If **\_DEBUG\_FORMAT** equals ISD, then **-g** is translated to **TEST** (the old translation).

For the impact on the runtime environment, see “Debug format and c89 -g flag option translation” on page 103.

For more information about using the c89 utility, see the c89 utility information in *z/OS XL C/C++ User's Guide*.

## Changes that affect use of the xlc utility

When you use the xlc utility to compile or link an existing application, be aware of the following potential migration issues:

- Changes in processing of return code (see “Exposure of build problems and xlc utility”)
- Changes in processing of source file comments (see “When C++ style comments are the default”)

### Exposure of build problems and xlc utility

As of z/OS V1R10 XL C/C++ compiler, the xlc utility handles the \*\_ACCEPTABLE\_RC environment variable as the c89 utility handles it. This permits users to specify acceptable return codes in order to expose the same build problems that are exposed with the c89 utility.

You will notice a change in behavior if:

- You use the xlc utility to compile source programs or link-edit object files in an environment in which the \*\_ACCEPTABLE\_RC environment variable is exported:
- The \*\_ACCEPTABLE\_RC environment variable has a value other than "4".

Otherwise, the xlc utility behaves the same as it did for earlier releases (assuming you do not use the acceptable\_rc configuration file attribute).

For detailed information about the \*\_ACCEPTABLE\_RC environment variable, see *z/OS UNIX System Services Command Reference*. For more information about specifying acceptable return codes, see *z/OS XL C/C++ User's Guide*.

### When C++ style comments are the default

As of z/OS V1R7 XL C/C++, the **xlc** command causes the compiler to generate C++ style comments by default. This change will not normally affect your program. But in the special cases where it does (as shown in the example below), you must either override **-qcplusplusmt** or change your source code.

In Figure 20, the intention is to increment the input by one.

```
printf("%d\n", i//*something*  
+1);
```

*Figure 20. C++ style comment*

Prior to z/OS V1R7 XL C/C++ compiler, the compiler saw the equivalent of: `printf("%d\n", i / +1);` and if the input is 4, the output is also 4.

As of z/OS V1R7 XL C/C++ compiler, the compiler sees the equivalent of: `printf("%d\n", i +1);` and if the input is 4, the output is 5, as intended.

---

## Changes that affect JCL procedures

Memory requirements for compilation may increase for successive releases as new logic is added. If you cannot recompile an application that you successfully compiled with a previous release of the compiler, try increasing the region size. For the current default region size, refer to the *z/OS XL C/C++ User's Guide*.

## User-defined conversion tables and iconv() functions

As of z/OS V1R9, the `iconv()` family of functions utilizes character conversion services provided by Unicode Services (UCS). Prior to z/OS V1R9 releases, the `iconv()` function used either a single byte or a double byte substitution character; single-byte and double-byte substitution characters were never mixed. As of z/OS V1R9, the `iconv()` function will use a single byte substitution character when converting single byte characters and a multibyte substitution character when converting multibyte characters in a mixed character set conversion. The environment variables, `_ICONV_MODE` and `_ICONV_TECHNIQUE` control function behavior.

These changes will affect your compilation only if both of the following conditions are true:

- Your JCL does specifies user-defined conversion tables.
- Your JCL uses conversion techniques other than LMREC (the default value for `_ICONV_TECHNIQUE`).

Otherwise, set the `_ICONV_MODE` environment variable to C in order to access the new UCS character conversion services.

**Note:** When Unicode Services are being used, the `_ICONV_UCS2` and `_ICONV_PREFIX` environment variables have no meaning.

The `iconv()` function returns the number of nonidentical conversions performed during a conversion. As of z/OS V1R9, the `iconv()` function interprets nonidentical conversion more strictly. This means that the nonidentical conversion count for the same input buffer contents might be higher than it was for compilations under previous releases.

If your program includes CICS statements, also see “Customized CEECCSD.COPY and CEECCSDX.COPY files and `iconv()` changes” on page 136.

**Note:** As of z/OS V1R11, IBM will no longer ship `uconvTable` binary tables in either the `installation-prefix.SCEEUTBL` data set or the z/OS UNIX file system directory `/usr/lib/nls/locale/uconvTable`.

## ILP32 compiler option and name mangling

As of z/OS V1R9, the default name mangling suboption under ILP32 is `zOSV1R2`, whether the ILP32 option is specified during the compiler invocation or used by default. Any JCL procedure that is run under the ILP32 compiler option (either explicitly or by default), and does not specify the suboption that controls the name mangling conventions, will instruct the compiler to mangle names differently that it did in earlier supported releases.

This change applies to batch processing only. For programs that are compiled under UNIX System Services, there is no change in behavior.

**Note:** In earlier supported releases, when ILP32 was either explicitly specified in the JCL or used by default, the default name mangling suboption was `ANSI` instead of `zOSV1R2`.

## IPA(LINK) compiler option and very large applications

As of z/OS V1R12 XL C/C++, when using the IPA compiler option to compile very large applications, you might need to increase the size of the work file

associated with SYSUTIP DD in the IPA Link step. If you are linking the application in a USS environment, you can control the size of this work file with the `_CCN_IPA_WORK_SPACE` environment variable. If particularly large source files are compiled with IPA, the default size of the compile-time work files might also need to be increased. These can be modified via the `prefix_WORK_SPACE` environment variables.

## IPA(LINK) compiler option and exploitation of 64-bit virtual memory

As of z/OS V1R12 XL C/C++, the compiler component that executes IPA at both compile and link time is a 64-bit application, which will cause an XL C/C++ compiler ABEND if there is insufficient storage. The default MEMLIMIT system parameter size in the SMFPRMxparmlib member should be at least 3000 MB for the link, and 512 MB for the compile. The default MEMLIMIT value takes effect whenever the job does not specify one of the following:

- MEMLIMIT in the JCL JOB or EXEC statement
- REGION=0 in the JCL

### Note:

- The compiler component that executes IPA(LINK) has been a 64-bit application since z/OS V1R8 XL C/C++ compiler.
- The MEMLIMIT value specified in an IEFUSI exit routine overrides all other MEMLIMIT settings.

The UNIX System Services `ulimit` command that is provided with z/OS can be used to set the MEMLIMIT default. For information, see *z/OS UNIX System Services Command Reference*. For additional information about the MEMLIMIT system parameter, see *z/OS MVS Programming: Extended Addressability Guide*/*z/OS MVS Programming: Extended Addressability Guide*.

As of z/OS V1R8 XL C++ compiler, the EDCI, EDCXI, EDCQI, CBCI, CBCXI, and CBCQI cataloged procedures, which are used for IPA Link, contain the variable IMEMLIM, which can be used to override the default MEMLIMIT value.

---

## JCL that runs pre-z/OS V1R5 C/C++ programs

As of z/OS V1R5, C++ compiler the CBCI and CBCXI procedures contain the variable CLBPRFX. If you have any JCL that uses these procedures, you must either customize these procedures (for example, at installation time) or modify your JCL to provide a value for CLBPRFX.

---

## Compiler options that manage Standard C++ compliance

To make an application conform to the currently supported Standard C++, you might need to change existing source code. You can use the compiler options and suboptions to manage those phases. For details, refer to Language element control options in *z/OS XL C/C++ User's Guide, SC14-7307*.

---

## Impact of recompiling applications that include <net/if.h> with the `_XOPEN_SOURCE_EXTENDED` feature test macro

As of z/OS V1R9, BSD-like socket definitions will not be automatically exposed when XPG 4.2 namespace is requested. To avoid violation of the standard UNIX namespace, the definitions are protected with the `_OPEN_SYS_IF_EXT` feature test macro.

**Note:** BSD sockets are used to manipulate network interfaces that are defined in <net/if.h>. For additional information about header files, see *z/OS XL C/C++ Runtime Library Reference, SC14-7314*.

---

## Impact of recompiling applications that include the `pselect()` interface

As of z/OS V1R11, recompilation of an existing XL C/C++ application that includes the <sys/select.h> header might fail if the application calls the `pselect()` interface and the undefined `_POSIX_C_SOURCE 200112L` feature test macro (or equivalent). If you need to recompile applications that call `pselect()`, you must define the `_POSIX_C_SOURCE` feature test macro (or equivalent) prior to including the system headers. Prior to z/OS V1R11, the `pselect()` declaration in <sys/select.h> was not protected by a feature test macro.

---

## Impact of recompiling with the `_OPEN_SYS_SOCKET_IPV6` macro

As of z/OS V1R7, recompiling an earlier C/C++ program that uses the `_OPEN_SYS_SOCKET_IPV6` feature test macro will expose new definitions in Language Environment header files. See “New definitions exposed by use of the `_OPEN_SYS_SOCKET_IPV6` macro” on page 82.

---

## Impact of recompiling code that relies on `math.h` to include IEEE 754 interfaces

As of z/OS V1R9 XL C/C++ compiler, recompilation of earlier C/C++ applications will fail if the code relies upon `math.h` to include `_Ieee754.h`. See “Potential need to include `_Ieee754.h`” on page 81.

---

## Chapter 16. Bind-time migration issues with earlier z/OS C/C++ programs

If you are relinking load modules or program objects from a previous release of z/OS C/C++ compiler, be aware of the following potential migration issues:

- “Unexpected "missing symbol" error (C++ only)”
- “Program modules from an earlier release”
- “Alignment incompatibilities between object models” on page 102
- “Alignment incompatibilities between XL C and XL C++ output with #pragma pack(2)” on page 102
- “Debug format and c89 -g flag option translation” on page 103
- “argc argv parsing support for Metal C programs” on page 103

---

### Unexpected "missing symbol" error (C++ only)

If the binder is generating "missing symbol" error messages that did not appear with earlier compilers, it might be due to the change in the treatment of the using directive that was introduced in the z/OS V1R10 XL C++ compiler. See “Unqualified name lookups and the using directive” on page 119.

---

### Program modules from an earlier release

When you use z/OS V2R1 XL C/C++ compiler to bind earlier program modules, be aware of the following migration issues:

- “Namespace pollution binder errors”
- “c89 COMPAT binder option default and programs from an earlier release” on page 102

### Namespace pollution binder errors

As of z/OS V1R8 XL C/C++ compiler, when you target OS/390 V2R10 or an earlier release while binding or linking your application, you might encounter the namespace pollution error shown in Figure 21.

**Note:** z/OS V1R1 C/C++ compiler is the same as OS/390 V2R10 C/C++ compiler. OS/390 V2R10 is also reshipped in z/OS V1R2 through to V1R6.

```
IEW2456E 9207 SYMBOL terminate_3stdFv UNRESOLVED. MEMBER COULD NOT BE INCLUDED
        FROM THE DESIGNATED CALL LIBRARY.
FSUM3065 The LINKEDIT step ended with return code 8.
```

*Figure 21. IEW2456E namespace pollution error*

If you encounter the error shown in Figure 21, use the code shown in Figure 22 on page 102 inside a header file that is included by the affected source.

```

#ifdef __cplusplus
#if ((__COMPILER_VER__ >= 0x41080000) && (__TARGET_LIB__ == 0x220A0000))
namespace std { void terminate(); }
#pragma map(std::terminate, "terminate__Fv")
#endif
#endif

```

**Note:** To prevent targeting an inappropriate release, guard the `#pragma map` statement with the `__TARGET_LIB__` macro.

*Figure 22. Header file code that handles IEW2456E error condition*

## c89 COMPAT binder option default and programs from an earlier release

As of z/OS V1R8 XL C/C++, the c89 utility no longer emits the default for the COMPAT binder option. This change prevents inadvertent attempts to use features that are not supported by the targeted release. It means that you have the option to obtain the binder defaults for the COMPAT option but you are not forced to override the c89 default when you bind applications intended to run on earlier releases. If you want to maintain the previous c89 utility behavior, you must do one of the following:

- Set the `_PVERSION` environment variable to a release earlier than z/OS V1R8 XL C/C++.
- Specify the COMPAT option on the command line. For example:  
`-w1,compat=curr.`

If you want to override the binder default for the COMPAT option using the C/C++ cataloged procedures, specify the desired COMPAT option in the BPARM proc variable.

**Note:** When the TARGET compiler option is used, binder features that are not supported by the targeted release should not be used. In previous releases of the z/OS C/C++ compiler, the default COMPAT option had to be overridden.

---

## Alignment incompatibilities between object models

As of z/OS V1R6, C/C++ compilers support the IBM object model as well as the compat object model. The IBM object model has a more complex layout than the compat object model. The more complex layout supports 64-bit processing as well as 31-bit processing.

The IBM object model is the default for 64-bit processing, which is specified by the LP64 compiler option. The compat object model is the default for 31-bit processing, which is specified by the ILP32 compiler option. Because each object model uses a different memory layout, C++ constructs that work under the compat object model might not work under the IBM object model.

For more information, refer to The z/OS 64-bit environment in *z/OS XL C/C++ Programming Guide*.

## Alignment incompatibilities between XL C and XL C++ output with #pragma pack(2)

An aggregate, which contains char data type members only, has a natural alignment of one byte. Typically, XL C retains the natural one-byte alignment.



However, when **#pragma pack(2)** is applied to an aggregate, its alignment increases to two bytes. If you are binding both XL C and XL C++ program modules, and both C and C++ program modules use **#pragma pack(2)**, there might be alignment incompatibilities.

See “Unexpected C++ output with #pragma pack(2)” on page 84.

---

## Debug format and c89 -g flag option translation

As of z/OS V1R6 C/C++, the environment variable `_DEBUG_FORMAT` can be used with the c89 utility to specify translation of the **-g** flag option for 31-bit compilations:

- If `_DEBUG_FORMAT` equals DWARF (the default), **-g** is translated to `DEBUG(FORMAT(DWARF))`.
- If `_DEBUG_FORMAT` equals ISD, then **-g** is translated to TEST (the old translation).

For the impact on specification of compiler options, see “Debug format specification” on page 96.

For detailed information about using the c89 utility, see the c89 in *z/OS XL C/C++ User's Guide*.

---

## argc argv parsing support for Metal C programs

As of z/OS V1R13, the `argc argv` parsing capability is added to Metal C programs. If your Metal C programs work with standard `argc` and `argv` arguments, the newly enabled parsing code generated by the compiler might cause problems.

If you use `argc` and `argv` in your `main()` function, you need to add `CBC.SCCNOBJ` dataset to the binder `SYSLIB` for the resolution of `CCNZINIT` and `CCNZTERM` routines (`CCNZQINI` and `CCNZQTRM` for LP64). The `CCNZINIT` and `CCNZTERM` routines need NAB established for their stack space. If you supply your own prolog and epilog for `main()`, you need to allocate 1K of extra space (2K for LP64) in addition to the DSA size suggested by the compiler in the global `SET` symbol `&CCN_DSASZ`.

For more information, see `ARGPARSE` | `NOARGPARSE` in *z/OS XL C/C++ User's Guide*, SC14-7307.



---

## Chapter 17. Runtime migration issues with earlier z/OS C/C++ applications

Runtime migration issues with earlier z/OS C/C++ programs result from changes in the Language Environment services, or in changes in functionality of runtime options.

Be aware of the following potential migration issues:

- “Earlier AMODE 64 applications”
- “Retention of previous runtime behavior” on page 106
- “Failure of authentication process” on page 106
- “Internationalization issues” on page 108
- “Changes in math library functions” on page 109
- “Changes in floating-point support” on page 110
- “Changes in allocation of VSAM control blocks” on page 111
- “Changes to st\_mode attribute of AF\_UNIX socket files” on page 111
- “Changes to strfmon() output” on page 112
- “Changes to structure t\_opthdr in xti.h” on page 112
- “Removal of conversion table source code” on page 112

---

### Earlier AMODE 64 applications

When you run earlier applications under AMODE 64, be aware of the following potential issues:

- “HEAPPOOLS runtime option no longer ignored in all AMODE 64 applications”

#### **HEAPPOOLS runtime option no longer ignored in all AMODE 64 applications**

As of z/OS V1R10, Language Environment services will not ignore the HEAPPOOLS runtime option when AMODE 64 applications specify it by using the `_CEE_RUNOPTS` environment variable.

In earlier Language Environment releases, when the HEAPPOOLS runtime option was specified via the `_CEE_RUNOPTS` environment variable, it was handled as follows:

- When an AMODE 64 application spawned an AMODE 31 process, the AMODE 64 application would ignore the HEAPPOOLS runtime option, but the AMODE 31 process would accept and propagate it.
- When an AMODE 31 application spawned an AMODE 64 process, the AMODE 31 application would accept the HEAPPOOLS runtime option, but the AMODE 64 process would ignore it.

---

### Customized runtime libraries

Language Environment improvements might necessitate changing the way you build your libraries.

For a list of Language Environment references, refer to “Bibliography” on page 153.

---

## Failure of authentication process

If a pre-z/OS V1R10 XL C/C++ application fails to authenticate any password strings, it might be because the maximum length of Pass\_MAX has increased from 8 bytes to 255 bytes.

You should confirm that there is no change in password authentication behaviour by existing applications that use the `getpass()` function.

---

## Retention of previous runtime behavior

When your program is using Language Environment services, you can use the ENVAR runtime option to specify the values of environment variables at execution time. You can use some environment variables to specify the original runtime behavior for particular items. The following setting specifies the original runtime behavior for the greatest number of items:

```
ENVAR("_EDC_COMPAT=32767")
```

Alternatively, you can add a call to the `setenv()` function, either in the CEEBINT High-Level Language exit routine or in your `main()` program. If you use CEEBINT only, you will need to relink your application. If you add a call to `setenv()` in the `main()` function, you must recompile the program and then relink your application. For more information, refer to `setenv()` in *z/OS XL C/C++ Runtime Library Reference, SC14-7314* and to Using environment variables in *z/OS XL C/C++ Programming Guide*.

## Unexpected output from `fprintf()` or `fscanf()`

As of z/OS V1R8, XL C/C++ supports decimal floating point size modifiers ("D", "DD", and "H") for the `fprintf` and `fscanf` families of functions. If a percent sign (%) is followed by one of these character strings, which had no meaning under previous releases of z/OS XL C/C++, the compiler could interpret the data as a size modifier. Treatment of this condition is undefined and the behavior could be unexpected.

For example, Table 18 shows the output, under different conditions, for the following statement:

```
printf("This results in a 10% Deduction.\n");
```

Table 18. Potential results of `printf("This results in a 10% Deduction.\n");`

Compiler release	Hardware	Result
z/OS V1R9 XL C/C++	Without the DFP facility.	EDC6259S This function is not supported running on hardware that does not have the Decimal Floating Point Facility installed.
z/OS V1R9 XL C/C++	With the DFP facility.	The following is written to stdout: This results in a 10 2.000000e-390duction.
Earlier z/OS C/C++	Any hardware.	The following is written to stdout: This results in a 10Deduction.

See “Required changes to `fprintf` and `fscanf` strings %D, %DD, and %H” on page 82.

## IEEE754 math functions

As of z/OS V1R9, certain IEEE754 `fdlibm` math functions are replaced by code written by IBM Research. Some of those were enhanced to improve performance and accuracy. The earlier versions are still available. See “Changes in math library functions” on page 109.

## Internal timing algorithm specification

As of z/OS V1R8 XL C/C++ compiler, the internal timing algorithm uses the `_EDC_PTHREAD_YIELD` environment variable setting to control the time at which the processor is released.

If you want to continue to use the previous internal timing algorithm, use the following command:

```
_EDC_PTHREAD_YIELD=-1
```

For information about `_EDC_PTHREAD_YIELD` and setting environment variables, see Using environment variables in *z/OS XL C/C++ Programming Guide, SC14-7315*.

For information about the `pthread_yield()` and `sched_yield()` functions, see *z/OS XL C/C++ Runtime Library Reference, SC14-7314*.

## Daylight saving time definition

If you are using a locale that has been customized with `LC_TOD`, you need to be aware that as of z/OS V1R9, the Language Environment default daylight saving time (that for the U.S. Eastern time zone) is changed.

To retain the earlier daylight saving time, use either of the following methods:

- If the `TZ` environment variable is defined, reset it to override the default time zone, which is the U.S. Eastern time zone. `TZ` is typically set (with the value that is defined in either the `/etc/environment` or `/etc/profile` files) when the system is started.
- Replace the values in the `time_t` structure with those saved from your earlier `time.h` header file.

**Note:** The `time.h` header file contains declarations of all timezone-related subroutines and externals, as well as the `tm` structure.

## Changes to the `putenv()` function and POSIX compliance

As of z/OS V1R5 C/C++, the function `putenv()` places the string passed to `putenv()` directly into the array of environment variables. This behavior assures compliance with the POSIX standard.

Prior to z/OS V1R5 C/C++, the string used to define the environment variable passed into `putenv()` was not added to the array of environment variables. Instead, the system copied the string into system-allocated storage.

To allow the POSIX-compliant behavior of `putenv()`, do nothing; it's now the default condition.

To restore the previous behavior of `putenv()`, follow these steps:

1. Ensure that the environment variable, `_EDC_PUTENV_COPY`, is available on your pre-z/OS V1R5 system.
2. Set the environment variable `_EDC_PUTENV_COPY` to "YES".

For additional information, see:

- *z/OS XL C/C++ Runtime Library Reference*
- `_EDC_PUTENV_COPY` in *z/OS XL C/C++ Programming Guide*

---

## Internationalization issues

If you are running an application that was last compiled under z/OS V1R2, z/OS V1R3, or z/OS V1R4, or z/OS V1R5, be aware of the following internationalization issues:

- “Default daylight saving time change”
- “EEC default currency update”
- “Movement of LOCALDEF utilities to new data sets”

### Default daylight saving time change

As of z/OS V1R9, the Language Environment default daylight saving time is changed. Functions that depend on the change to or from daylight saving time will be executed in accordance with the new default. For example, a function such as `localtime()` will use the new default daylight saving time to return the local time.

If you are using a locale that has been customized with the LC\_TOD IBM extension, you can retain the previous daylight saving time. See “Daylight saving time definition” on page 107.

**Note:** The LC\_TOD IBM extension specifies the rules used to define the beginning, end, and duration of daylight savings time, and the difference between local time and Greenwich Mean Time.

### EEC default currency update

Prior to z/OS V1R6, the default currency for EEC was set to local currency in the LC\_MONETARY category of the locale. If you wanted to set Euro as currency, the `@euro` locales would need to be set using `setlocale()`.

As of z/OS V1R6, the LC\_MONETARY information in the base locale is now preset to use the Euro, which means that the Euro is the default currency. If you want your applications to continue using the old (local) currency, you will need to issue `setlocale()` with the new `@preeuro` locale as the parameter.

Behavior of the current `@euro` locales has not changed.

For z/OS V1R7 to z/OS V1R9, Venezuela is changing its currency from bolivar to bolivar fuerte. The national currency symbol changes from Bs to BSF, and the international currency symbol changes from VEB to VEF. If you want to keep using the old currency symbols, the Bs or VEB (bolivar), you must use `setlocale()` with a locale name of "Es\_VEO" for the language-territory part, instead of "Es\_VE".

As of z/OS V1R9, Malta is adopting the euro currency. If you want to keep using the old currency symbol, you must use the `@preeuro` locales.

### Movement of LOCALDEF utilities to new data sets

As of z/OS V1R6, the following LOCALDEF utilities have been moved to new data sets.

Utility	From C/C++ data set	To Language Environment data set
LOCALDEF	CBC.SCCNUTL	CEE.SCEECLST
EDCLDEF	CBC.SCCNPRC	CEE.SCEEPROC
EDCXLDEF	CEE.SCCNPRC	CEE.SCEEPROC
CCNELDEF	CBC.SCCNCMP	CEE.SCEERUN2
CCNLMSG	CBC.SCCNCMP	CEE.SCEERUN2

If you use the MVS batch or TSO localedef (LOCALDEF) utility interfaces, you might need to do the following:

- Add or replace the Language Environment procedures library (CEE.SCEEPROC) where you currently have the C/C++ procedures library (CBC.SCCNPRC).
- Add or replace the Language Environment clist/exec library (CEE.SCEECLST) where you currently have the C/C++ clist/exec library (CBC.SCCNUTL). In addition, you may need to customize the Language Environment customization member (CEE.SCEECLST(CEE.CEL4CUST)) in addition to customizing the C/C++ customization member (CBC.SCCNUTL(CBC.CCNCCUST)).
- Add the Language Environment library CEE.SCEERUN2 (in addition to CEE.SCEERUN) where you currently have the C/C++ library CBC.SCCNCMP.

---

## Changes in math library functions

As of z/OS V1R9, certain IEEE754 fdlibm math functions are replaced by code written by IBM Research.

The earlier versions of functions that are more closely aligned with the C99 standard are no longer available. Neither the `_IEEEV1_COMPATIBILITY` feature test macro nor the `_EDC_IEEEV1_COMPATIBILITY` environment variable can be used to affect these functions.

The earlier versions of functions with performance and accuracy enhancements are still available. See Table 19 on page 110.

To use earlier versions of the IEEE754 fdlibm math functions, use either of the following methods:

- When using the `FLOAT(IEEE)` compiler option, use the `_IEEEV1_COMPATIBILITY` feature test macro.
- When variable mode is in effect, use environment variable `_EDC_IEEEV1_COMPATIBILITY_ENV=ON`.

**Note:** Variable mode is in effect under either of the following conditions:

- The `_FP_MODE_VARIABLE` feature test macro is used.
- The `math.h` header file is not included.

To modify your source code to use the new performance and accuracy enhancements, use the information in Table 19 on page 110.

Table 19. IEEE754 fdlibm math functions replaced in z/OS V1R9 XL C/C++

Math functions that are enhanced for performance and accuracy	Math functions that are replaced but still available
acos()	acosl()
acosh()	asinl()
asin()	atanl()
asinh()	atan2l()
atan()	coshl()
atanh()	cosl()
atan2()	frexpl()
cbrt()	ldexpl()
cos()	log10l()
cosh()	modfl()
erf()	powl()
erfc()	sinhl()
exp()	tanl()
expm1()	tanhl()
gamma()	
hypot()	
lgamma()	
log()	
log1p()	
log10()	
pow()	
rint()	
sin()	
sinh()	
tan()	
tanh()	

## Changes in floating-point support

Changes in hexadecimal floating-point support could produce unexpected results.

### Hexadecimal floating-point notation

Changes in support of hexadecimal floating point notation in the numeric conversion functions introduced in *Programming languages - C (ISO/IEC 9899:1999)* can alter the behavior of well-formed applications that comply with the *Programming languages - C (ISO/IEC 9899:1990)* standard and earlier versions of the base documents. One such example would be:



```

int what_kind_of_number (char *s){
    char *endp; *EXP = "p+0"
    double d;
    long l;

    d = strtod(s,&endp);
    if (s != endp && *endp == '\0')
        printf("It is a float with value %g\n", d);           1
    else{
        l = strtol(s,&endp,0);
        if (s != endp && (strcmp(endp,EXP)== 0))
            printf("It is an integer with value %ld\n", l);   2
        else
            return 1;
    }
    return 0;
}

```

**Notes:**

1. If the function is called with: `what_kind_of_number ("0xAp+0")` and the runtime library is C99-compliant, the output is: It is a float with value 10.
2. If the function is called with: `what_kind_of_number ("0xAp+0")` and the runtime library is not C99-compliant, the output is: It is an integer with value 10 and an exception is raised.

*Figure 23. Example of how C99 changes in hexadecimal floating-point notation affect well-formed code*

## Floating-point special values

The numeric conversion functions accept the following special values at all times:

- `±inf` or `±INF`
- `±nanq` or `±nanq(n-char-sequence)`, and `±NANQ` or `±NANQ(n-char-sequence)`
- `±nans` or `±nans(n-char-sequence)`, and `±NANS` or `±NANS(n-char-sequence)`
- `±nan` or `±nan(n-char-sequence)`, and `±NAN` or `±NAN(n-char-sequence)`

**Note:** Neither the z/OS XL C/C++ compiler nor the Language Environment C/C++ runtime library includes `_Imaginary` or formal support of the IEC 60559 floating point as described in Annex F and Annex G of the C99 standard.

## Changes in allocation of VSAM control blocks

As of z/OS V1R10, the XL C/C++ compiler instructs VSAM, by default, to allocate control blocks and I/O buffers above the 16-MB line.

If you determine that this change could be causing a problem, you can use the VSAM JCL parameter AMP to override the default.

## Changes to `st_mode` attribute of AF\_UNIX socket files

As of z/OS V2R1, the retrieved file type of AF\_UNIX socket files that are returned in `st_mode` is `S_IFSOCK`, rather than `S_IFCHR`. Functions `stat()`, `lstat()`, `stat_o()`, `lstat_o()`, and `__readdir2()` are affected.

You must examine programs that use the affected functions and check the type of AF\_UNIX socket files to ensure compatibility with the updated function behavior.

---

## Changes to strfmon() output

As of z/OS V2R1, the alignment of formatted output from `strfmon()` is changed. When `#n` and `(` are specified in the input of `strfmon()`, the formatted output of positive and negative values are aligned in the same columns, as required by the UNIX Standard. This causes the output of a positive value to be wider than in previous releases.

For example, the input format of `strfmon()` is `%(#5n`, which specifies that 5 digits are expected to be formatted to the left of the radix character and that negative amounts are enclosed with parentheses. Given a positive value `1234.56` and a negative value `-1234.56`, the output of `strfmon()` is as follows:

```
[ 123456 ]  
[( 123456)]
```

---

## Changes to structure t\_opthdr in xti.h

As of z/OS V2R1, the member type of structure `t_opthdr` is changed from unsigned `int` to unsigned `long`, when not compiling with `AMODE 64`.

Programs that are compiled before this change can still run correctly without being re-compiled. Warning messages about conversion between unsigned `int` and unsigned `long` might be reported at compile time if a program does not comply with the new version of structure `t_opthdr`.

---

## Changes to getting group or user database entry

As of z/OS V2R1, the case of not found database entry is not treated as an error case. As required by the UNIX standard, when the group or user database entry that is associated with the specified name or ID is not found, the calling function will not set `errno`. The impacted functions are `getgrnam()`, `getpwnam()`, and `getgrgid()`.

To ensure compatibility with the updated behavior, examine your programs that get database entry by calling changed functions.

---

## Removal of conversion table source code

As of z/OS V1R12, the C/C++ runtime library will no longer ship any `ucmap` source code or `genxlt` source code for character conversions now being performed by Unicode Services.

Users with customized conversion tables should now generate custom Unicode Services conversion tables.

Users of the `iconv()` family of functions testing to a "known conversion result" who experience testcase failures need to update their expected results to the new conversion results.

Users wanting to create custom conversion tables involving any of the `CCSIDs` related to the conversion table source no longer being shipped should now generate custom Unicode Services conversion tables instead of custom Language Environment conversion tables.

The `<INSTALLATION_PREFIX>.SCEEUMAP` data set will no longer be shipped.

The /usr/lib/nls/locale/ucmap HFS directory will no longer be shipped.

**Note:** The `_ICONV_TECHNIQUE` environment variable must be set to the same technique search order value used for the customized Unicode Services table in order for the `iconv()` family of functions to use the customized Unicode Services table. For example, if you want the `iconv()` family of functions to use a user-defined Unicode Services table with a technique search order of 2, the `_ICONV_TECHNIQUE` environment variable should be set to `2LMREC`.

For information about how to generate and use custom Unicode Services conversion tables, see *Support for Unicode: Using Unicode Services*, SA22-7649.



---

## Part 5. ISO Standard C++ compliance migration issues

*Programming languages - C++ (ISO/IEC 14882:2003(E))* documents the currently supported Standard C++.

As of z/OS V1R2 C++, the z/OS C++ compiler was compliant with *Programming languages - C++ (ISO/IEC 14882:1998(E))*.

As of z/OS V1R7 XL C/C++:

- z/OS C++ was compliant with *Programming languages - C++ (ISO/IEC 14882:2003(E))*.
- OS/390 V2R10 compiler was no longer shipped with the z/OS product. This means that programs compiled with the z/OS C++ compiler must be compliant with *Programming languages - C++ (ISO/IEC 14882:2003(E))* or *Programming languages - C++ (ISO/IEC 14882:1998(E))*.

**Note:** You can determine the ISO Standard level that is supported by the compiler by checking the standard macro `__cplusplus` and its value, which remains unchanged from z/OS V1R6 C++. This macro has the value 199711. If you are compiling a C++ translation unit, the name `__cplusplus` is defined to the value 199711L.

The following topics discuss the implications of migrating applications that were created with C++ compilers that are not compliant with *Programming languages - C++ (ISO/IEC 14882:2003(E))*

- Chapter 18, "Language level and your Standard C++ compliance objectives," on page 117
- Chapter 19, "Changes that affect Standard C++ compliance of language features," on page 119



## Chapter 18. Language level and your Standard C++ compliance objectives

Code that compiles without errors in pre-z/OS C++ V1R2 compilers might produce warnings or error messages in the z/OS V2R1 XL C++ compiler. This could be due either to changes in the language or to differences in the compiler behavior. Language elements that may affect your code are shown in Chapter 19, “Changes that affect Standard C++ compliance of language features,” on page 119.

Table 20 shows the Standard C++ migration objectives and the recommended approach for each.

**Note:** Full conformance can be achieved gradually by migrating to selected individual language features in phases.

Table 20. Standard C++ migration objectives and approaches

Is code compliant with 1998 ISO Standard C++?	Compliance objective	Action
Yes (ported or new).	Migrate to the 2003 Standard C++.	No action required.
	Remain compliant with 1998 Standard C++.	Use one of the following compiler options and suboptions: <ul style="list-style-type: none"> <li>LANGLVL(ANSI)</li> <li>LANGLVL(STRICT98)</li> </ul> <b>Notes:</b> <ol style="list-style-type: none"> <li>LANGLVL(ANSI) and LANGLVL(STRICT98) are synonymous.</li> <li>You can use compiler options to control individual language features. See the "Compatibility options for z/OS XL C/C++ compiler" table in the LANGLVL description, <i>z/OS XL C/C++ User's Guide, SC14-7307</i>.</li> </ol>
No	Use Standard C++ language features, even if code must be modified.	Use the following compiler options and suboptions to aid the migration process: <ul style="list-style-type: none"> <li>LANGLVL(COMPAT92) if your code compiles with a previous compiler and you want to move to z/OS V2R1 XL C/C++ with minimal changes. <b>Note:</b> This group is the closest you can get to the behavior of the previous compilers.</li> <li>For information about compiler suboptions that you can use to control individual language features, refer to "Compatibility options for z/OS(R) XL C/C++ compiler" in the LANGLVL compiler option description in <i>z/OS XL C/C++ User's Guide, SC14-7307</i>.</li> </ul>
	Avoid modifying code and ignore Standard C++ language features.	Use LANGLVL(COMPAT92) to tolerate language incompatibilities.





---

## Chapter 19. Changes that affect Standard C++ compliance of language features

For information about setting the language level to meet your Standard C++ compliance objectives, see Chapter 18, “Language level and your Standard C++ compliance objectives,” on page 117.

Refer to the *z/OS XL C/C++ Language Reference, SC14-7308* for details.

---

### Unqualified name lookups and the using directive

As of z/OS V1R10 XL C++ compiler, the location of the using directive determines how function calls are resolved.

Figure 24 provides an example of code that will be compiled differently by z/OS V1R10 XL C++ compiler than it was by earlier XL C++ compilers.

```
}namespace bb {  
  double sp1(double) { return 1.0; }  
}  
  
int main()  
{  
  double sp1(double);  
  sp1(0);  
  return 0;  
}  
using namespace bb;
```

*Figure 24. Example of code with a using directive*

Prior to z/OS V1R10 XL C++ compiler, the compiler would resolve the call to the function `sp1` in the namespace `bb` even though the statement `using namespace bb;` is not located before the function is called inside the main routine.

In the example in Figure 24, the declaration of `sp1` in the main function is a declaration in the global namespace. As of z/OS V1R10 XL C++ compiler, the compiler will resolve that function call to the declaration in the global namespace. Because the definition of `sp1` is missing in the global namespace, the binder will generate an error message.

To avoid the error at bind time, you can modify the example in Figure 24 in any of the following ways:

- Explicitly resolve the function call to `sp1` in the namespace `bb` by using the namespace qualifier in the function call
- Implicitly resolve the function call to `sp1` in the namespace `bb` by moving the using directive above the main routine.
- Make the function definition available in the global namespace.

For detailed information, refer to The using declaration and namespaces in *z/OS XL C/C++ Language Reference, SC14-7308*.

For examples of the using directive in a sample program, see `CCNUBRC` and `CLB3ATMP.CPP`. These are documented in *z/OS XL C/C++ User's Guide*.

---

## Order of destruction for statically initialized objects

As of z/OS V1R5 C++ compiler, you can use the `LANGLVL(NOANSISINIT)` option to maintain the order of destruction for statically initialized objects whenever you compile programs that had previously been compiled with z/OS V1R1 and earlier C++ compilers.

As of z/OS V1R2 C++ compiler, DLLs built by the compiler run object destructors differently from those created with the earlier C++ compilers.

**Note:** The compiler became fully compliant with the C++ 2003 standard as of z/OS V1R2 C++ compiler.

*Table 21. Destruction of statically initialized objects and compliance with Standard C++*

z/OS V1R1 and earlier C++ compilers	z/OS V1R2 and later compilers
Destructor calls are run as the last thing on the <code>atexit</code> list, as part of the termination code.	For objects created with the Standard C++ way of initializing ( <code>LANGLVL(ANSISINIT)</code> ): <ul style="list-style-type: none"><li>• Destructor calls for objects created by z/OS V1R2 and later compilers are added to the <code>atexit</code> list. This list will then be run before the <code>atexit</code> entry for the termination code.</li><li>• Any DLL built with z/OS V1R2 and later compilers will have the destructors for the global objects run in the wrong order relative to other DLLs or main program that were built with z/OS V1R1 and earlier C++ compilers.</li></ul>

---

## Implicit integer type declarations

The use of an implicit `int` in a declaration, as shown in Figure 25, does not comply with Standard C++. If you need to comply with the Standard C++, specify the type of every function and variable. Otherwise, use the `LANGLVL(IMPLICITINT)` option to compile code containing declarations of implicit integer types.

```
const i; // previously meant const int i
main() { } // previously returned int
```

*Figure 25. Declaration of implicit integer type.*

As of z/OS V1R2 C++, the following code is no longer valid:

```
inline f() {
    return 0;
}
```

---

## Scope of for-loop initializer declarations

In Standard C++, a variable in a for loop initializer declaration is declared within, and scoped to, the loop body.

If you are migrating a program that was last compiled by a pre-z/OS V1R2 C++ compiler, you should be aware that such variables were declared outside of the for-loop, and were scoped to the lexical block containing the for-loop. See Figure 26 on page 121.

As of z/OS V1R2 C++ compiler, you can retain the original scope of a for-loop initializer declaration by specifying the LANGLVL(NOANSIFOR) compiler option.

```
int i=0;

void f() {
    for(int i=0; i<10; i++) {
        if(...) break;
    }
    if(i==10) { ... }    // 1
    ...
}
```

**Note:** Prior to z/OS V1R2, the variable `i` was declared outside the for-loop.

*Figure 26. A for-loop initializer declaration that does not comply with Standard C++*

---

## Visibility of friend declarations

As of the z/OS V1R2 C++ compiler, a friend class is not visible unless it is introduced into scope by another declaration, as shown in Figure 27. To allow friend declarations without elaborated class names, use the LANGLVL(OLDFRIEND) option.

```
class C {
    friend class D;
};
D* p; // error, D not in scope
```

*Figure 27. friend declaration that is not visible*

A friend class declaration must always be elaborated, as shown in Figure 28.

```
friend class C; // need class keyword
```

*Figure 28. friend declaration that is made visible.*

---

## Migration of friend declarations in class member lists

A friend declaration in a class member list grants, to the nominated friend function or class, access to the private and protected members of the enclosing class. In pre-z/OS V1R2 C++ compilers, friend declarations introduce the name of a nominated friend function to the scope that encloses the class containing the friend declaration. As of z/OS V1R2 C++ compiler, friend declarations do not introduce the name of a nominated friend function to the scope that encloses the class containing the friend declaration.

The code in Figure 29 on page 122 will not compile successfully because the z/OS V2R1 XL C/C++ compiler will not know the function name `lib_func1` at the point at which it is called in the function `f`.

```

// g.C
// ---
class A {
    friend int lib_func1(int); // This function is from a library.
};
❶
int f(){
    return lib_func1(1);
}

```

**Note:** The code in Figure 29 will compile successfully if the following declaration is added to the file in the global namespace scope at some point prior to the definition of the function named `f`:

```
int lib_func1(int);
```

Figure 29. Example of code that does not introduce a friend function

---

## cv-qualifications when the thrown and caught types are the same

As of z/OS V1R2 C++ compiler:

- A temporary copy is thrown rather than the actual object itself.
- The cv-qualification in the catch clause is not considered when one of the following are true:
  - The type caught is the same (possibly cv-qualified) type as that thrown.
  - The type caught is a reference to the same (possibly cv-qualified) type.

**Note:** *cv* is short form for *const/volatile*.

- New casts also throw exceptions.

This is not the case in z/OS V1R1 and earlier C++ compilers. As of z/OS V1R5 C++ compiler, there is no available option to enable pre- z/OS V1R2 behavior.

---

## Compiler options that are introduced in C++11 standard

The following topics describe compiler options that are introduced in the C++11 standard as of z/OS V2R1 XL C++ compiler. To make an application conform to the currently supported C++11 standard, you might need to change your existing source code.

- “LANGLVL(AUTOTYPEDEDUCTION) compiler option (C++11)” on page 123
- “LANGLVL(C1XNORETURN) compiler option (C++11)” on page 123
- “LANGLVL(C99LONGLONG) compiler option (C++11)” on page 123
- “LANGLVL(C99PREPROCESSOR) compiler option (C++11)” on page 123
- “LANGLVL(CONSTEXPR) compiler option (C++11)” on page 124
- “LANGLVL(DECLTYPE) compiler option (C++11)” on page 124
- “LANGLVL(DEFAULTANDDELETE) compiler option (C++11)” on page 124
- “LANGLVL(DELEGATINGCTORS) compiler option (C++11)” on page 124
- “LANGLVL(EXPLICITCONVERSIONOPERATORS) compiler option (C++11)” on page 124
- “LANGLVL(EXTENDED0X) compiler option” on page 94
- “LANGLVL(EXTENDED FRIEND) compiler option (C++11)” on page 124
- “LANGLVL(EXTENDEDINTEGERSAFE) compiler option (C++11)” on page 125

- “LANGLVL(EXTERNTEMPLATE) compiler option (C++11)” on page 125
- “LANGLVL(INLINENAMESPACE) compiler option (C++11)” on page 125
- “LANGLVL(REFERENCECOLLAPSING) compiler option (C++11)” on page 125
- “LANGLVL(RIGHTANGLEBRACKET) compiler option (C++11)” on page 125
- “LANGLVL(RVALUEREFERENCES) compiler option (C++11)” on page 126
- “LANGLVL(SCOPEDENUM) compiler option (C++11)” on page 126
- “LANGLVL(STATIC\_ASSERT) compiler option (C++11)” on page 126
- “LANGLVL(VARIADICTEMPLATES) compiler option (C++11)” on page 126
- “WARN0X compiler option (C++11)” on page 126

**Note:** C++11 is a new version of the C++ programming language standard. IBM continues to develop and implement the features of the new standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the features of the C++11 standard is complete, including the support of a new C++ standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new features of the C++11 standard and therefore they should not be relied on as a stable programming interface.

### **LANGLVL(AUTOTYPEDEDUCTION) compiler option (C++11)**

This option controls whether the auto type deduction feature is enabled. When LANG(AUTOTYPEDEDUCTION) is in effect, you do not need to specify a type when declaring a variable. Instead, the compiler deduces the type of an auto variable from the type of its initializer expression. The default is LANG(NOAUTOTYPEDEDUCTION). For detailed information, see **AUTOTYPEDEDUCTION | NOAUTOTYPEDEDUCTION** that is documented in *z/OS XL C/C++ User's Guide*.

### **LANGLVL(C1XNORETURN) compiler option (C++11)**

This option controls whether the `_Noreturn` function specifier is supported. The default is LANGLVL(NOC1XNORETURN). For detailed information, see **C1XNORETURN | NOC1XNORETURN** that is documented in *z/OS XL C/C++ User's Guide*.

### **LANGLVL(C99LONGLONG) compiler option (C++11)**

This option controls whether the feature of C99 long long with IBM extensions adopted in C++11 is enabled. When LANG(C99LONGLONG) is in effect, the C++ compiler provides the C99 long long with IBM extensions feature. Source compatibility between the C and the C++ language is improved. The default is LANG(NOC99LONGLONG). For detailed information, see **C99LONGLONG | NOC99LONGLONG** that is documented in *z/OS XL C/C++ User's Guide*.

### **LANGLVL(C99PREPROCESSOR) compiler option (C++11)**

This option controls whether the C99 preprocessor features adopted in C++11 are enabled. When LANG(C99PREPROCESSOR) is in effect, C99 and C++11 compilers provide a common preprocessor interface, which can ease the porting of C source files to the C++ compiler and avoid preprocessor compatibility issues. The default is LANG(NOC99PREPROCESSOR). For detailed information, see **C99PREPROCESSOR | NOC99PREPROCESSOR** that is documented in *z/OS XL C/C++ User's Guide*.

## **LANGLVL(CONSTEXPR) compiler option (C++11)**

This option controls whether the generalized constant expressions feature is enabled. When you specify the `LANGLVL(CONSTEXPR)` option, the compiler extends the expressions permitted within constant expressions. A constant expression is one that can be evaluated at compile time. The default option is `LANGLVL(NOCONSTEXPR)`. For detailed information, see **CONSTEXPR** | **NOCONSTEXPR** that is documented in *z/OS XL C/C++ User's Guide*.

## **LANGLVL(DECLTYPE) compiler option (C++11)**

This option controls whether the declaration type feature is enabled. When `LANG(DECLTYPE)` is in effect, you can get a type that is based on the resultant type of a possibly type-dependent expression. The default is `LANG(NODECLTYPE)`. For detailed information, see **DECLTYPE** | **NODECLTYPE** that is documented in *z/OS XL C/C++ User's Guide*.

## **LANGLVL(DEFAULTANDDELETE) compiler option (C++11)**

This option controls whether the defaulted and deleted functions feature is enabled. With this feature, you can define explicitly defaulted functions whose implementations are generated by the compiler to achieve higher efficiency. You can also define deleted functions whose usages are disabled by the compiler to avoid calling unwanted functions. The default is `LANGLVL(NODEFAULTANDDELETE)`. For detailed information, see **DEFAULTANDDELETE** | **NODEFAULTANDDELETE** that is documented in *z/OS XL C/C++ User's Guide*.

## **LANGLVL(DELEGATINGCTORS) compiler option (C++11)**

This option controls whether the delegating constructors feature is enabled. When `LANG(DELEGATINGCTORS)` is specified, you can concentrate common initializations and post initializations in one constructor, which improves the readability and maintainability of the program. The default is `LANG(NODELEGATINGCTORS)`. For detailed information, see **DELEGATINGCTORS** | **NODELEGATINGCTORS** that is documented in *z/OS XL C/C++ User's Guide*.

## **LANGLVL(EXPLICITCONVERSIONOPERATORS) compiler option (C++11)**

This option controls whether the explicit conversion operators feature is enabled. When you specify the `LANGLVL(EXPLICITCONVERSIONOPERATORS)` option, you can apply the explicit function specifier to the definition of a user-defined conversion function, and thus to inhibit unintended implicit conversions through the user-defined conversion function. The default is `LANG(NOEXPLICITCONVERSIONOPERATORS)`. For detailed information, see **EXPLICITCONVERSIONOPERATORS** | **NOEXPLICITCONVERSIONOPERATORS** that is documented in *z/OS XL C/C++ User's Guide*.

## **LANGLVL(EXTENDED FRIEND) compiler option (C++11)**

Extended friend declarations which relax syntax rules governing friend declarations are supported by the new standard C++11. This feature is enabled by the new `LANGLVL(EXTENDED FRIEND)` compiler option, which can also be enabled by the group option `LANGLVL(EXTENDED0X)`. Otherwise, the feature is disabled by `LANGLVL(NOEXTENDED FRIEND)`. The default is `LANGLVL(NOEXTENDED FRIEND)`.

As of z/OS V1R11, when either `LANGLVL(EXTENDED_FRIEND)` or `LANGLVL(EXTENDED0X)` compiler option is turned on, the `__IBMCPP_EXTENDED_FRIEND` macro is defined with the value '1' by the compiler, and is undefined otherwise. For detailed information, see `EXTENDED_FRIEND` | `NOEXTENDED0XFRIEND` that is documented in *z/OS XL C/C++ User's Guide*.

### **LANGLVL(EXTENDEDINTEGERSAFE) compiler option (C++11)**

With this option, if a decimal integer literal that does not have a suffix containing `u` or `U` cannot be represented by the long long int type, you can decide whether to use the unsigned long long int to represent the literal or not. The default is `LANG(NOEXTENDEDINTEGERSAFE)`. For detailed information, see `EXTENDEDINTEGERSAFE` | `NOEXTENDEDINTEGERSAFE` that is documented in *z/OS XL C/C++ User's Guide*.

### **LANGLVL(EXTERNTEMPLATE) compiler option (C++11)**

Explicit instantiation declarations provide you with the ability to suppress implicit instantiations of a template specialization or its members when the `LANGLVL(EXTERNTEMPLATE)` option is turned on. It can also be enabled by the group options `LANGLVL(EXTENDED)` or `LANGLVL(EXTENDED0X)`. This feature is disabled when `LANGLVL(NOEXTERNTEMPLATE)` is set. The default is `LANGLVL(EXTERNTEMPLATE)`.

As of z/OS V1R11, when `LANGLVL(EXTERNTEMPLATE)` is set, the macro `__IBMCPP_EXTERN_TEMPLATE` is defined as the preprocessing number 1, and is undefined otherwise. In both cases, the macro is protected and a compiler warning will be emitted if it is undefined or redefined. For detailed information, see `EXTERNTEMPLATE` | `NOEXTERNTEMPLATE` that is documented in *z/OS XL C/C++ User's Guide*.

### **LANGLVL(INLINENAMESPACE) compiler option (C++11)**

This option controls whether the inline namespace definitions are enabled. A namespace definition preceded by an initial inline keyword is defined as an inline namespace. When `LANG(INLINENAMESPACE)` is in effect, members of the inline namespace can be defined and specialized as if they were also members of the enclosing namespace. The default is `LANG(NOINLINENAMESPACE)`. For detailed information, see `INLINENAMESPACE` | `NOINLINENAMESPACE` that is documented in *z/OS XL C/C++ User's Guide*.

### **LANGLVL(REFERENCECOLLAPSING) compiler option (C++11)**

This option controls whether the reference collapsing feature is enabled. The default option is `LANGLVL(NOREFERENCECOLLAPSING)`. For detailed information, see `REFERENCECOLLAPSING` | `NOREFERENCECOLLAPSING` that is documented in *z/OS XL C/C++ User's Guide*.

### **LANGLVL(RIGHTANGLEBRACKET) compiler option (C++11)**

This option controls whether the right angle bracket feature is enabled. The default option is `LANGLVL(NORIGHTANGLEBRACKET)`. For detailed information, see `RIGHTANGLEBRACKET` | `NORIGHTANGLEBRACKET` that is documented in *z/OS XL C/C++ User's Guide*.

## **LANGLVL(RVALUEREFERENCES) compiler option (C++11)**

This option controls whether the rvalue references feature is enabled. The default option is `LANGLVL(NORVALUEREFERENCES)`. For detailed information, see `RVALUEREFERENCES` | `NORVALUEREFERENCES` that is documented in *z/OS XL C/C++ User's Guide*.

## **LANGLVL(SCOPEENUM) compiler option (C++11)**

This option controls whether the scoped enumeration feature is enabled. The default option is `LANGLVL(NOSCOPEENUM)`. For detailed information, see `SCOPEENUM` | `NOSCOPEENUM` that is documented in *z/OS XL C/C++ User's Guide*.

## **LANGLVL(STATIC\_ASSERT) compiler option (C++11)**

This option controls whether the static assertions feature is enabled. When `LANGLVL(STATIC_ASSERT)` is set, a severe error message for compile-time assertions is issued on failure. The default is `LANG(NOSTATIC_ASSERT)`. For detailed information, see `STATIC_ASSERT` | `NOSTATIC_ASSERT` that is documented in *z/OS XL C/C++ User's Guide*.

## **LANGLVL(VARIADICTEMPLATES) compiler option (C++11)**

This option controls whether the variadic templates feature is enabled. When `LANGLVL(VARIADICTEMPLATES)` is set, you can define class and function templates that have any number (including zero) of parameters. The default is `LANG(NOVARIADICTEMPLATES)`. For detailed information, see `VARIADICTEMPLATES` | `NOVARIADICTEMPLATES` that is documented in *z/OS XL C/C++ User's Guide*.

## **WARN0X compiler option (C++11)**

The compiler option `WARN0X` controls whether to inform users with messages about differences in their programs caused by the migration from C++98 standard to C++11 standard. The default is `NOWARN0X`. For detailed information, see `WARN0X` | `NOWARN0X` that is documented in *z/OS XL C/C++ User's Guide*.

---

## **Errors due to changes in compiler behavior**

This topic describes coding that compiles without errors in *z/OS V1R1* and earlier C/C++ compilers but produces errors or warnings as of *z/OS V1R7 XL C/C++* compiler. For more details on compiler messages, refer to *z/OS XL C/C++ Messages, GC14-7305*.

### **C++ class access errors**

If your code has not been updated since *z/OS V1R2*, compiling it could raise exceptions because of changes in Standard C++ compliance. See “`CCN5413` exception” and “`CCN5193` exception” on page 127.

#### **CCN5413 exception**

An access specifier determines the accessibility of members that follow it, either until the next access specifier or until the end of the class definition. Violation of this rule will result in the following error message:

```
CCN5413:"A::B" is already declared with a different access
```

If you later define a class member within its class definition, its access specification must be the same as its declaration. The code in Figure 30 on page 127 violates this



rule.

```
class A {
    public:
        class B;
        private:
            class B {};    1
};
```

**Note:** The compiler will not allow the definition of class B because this class has already been declared as private. To correct the program, remove the private keyword.

*Figure 30. Code that results in CCN5413 exceptions*

### CCN5193 exception

When you specify a friend within a class, you must use the class name instead of the type-definition name. Without modification, the code in Figure 31 would result in the following error message:

CCN5193: A typedef name cannot be used in this context

```
class A { };
typedef A B;
class C {
    friend class B;    1
};
```

**Note:** Do not use the type-definition name; instead, use the name of the class:  
friend class A;

*Figure 31. Example: Correcting a type-definition name used out of context*

## Exceptions caused by ambiguous overloads

*Programming languages - C (ISO/IEC 9899:2003)* introduced error messages for standard floating point and long double overloads of standard math functions.

As of z/OS V1R2 C++ compiler, compiling the code in Figure 32 will produce the following error message:

CCN5219: The call to "pow" has no best match

To handle the exception, you could specify the LANGLVL(OLDMATH) option, which removes the float and long double overloads. If you don't want to remove the overloads, you can modify the code by casting the pow arguments.

```
#include <math.h>
int main()
{
    float a = 137;
    float b;
    b = pow(a, 2.0);    1
    return 0;
}
```

**Note:** The call to pow has no best match. To fix the problem, cast 2.0 to be of type float:  
b = pow(a, (float)2.0);

*Figure 32. Code modification to handle CCN5219 exception*

## Exceptions caused by user-defined conversions

User-defined conversions must be unambiguous, or they are not called.

```
//e.C
struct C {};
struct A {
    A();
    A(const C &);
    A(const A &);
};
struct B {
    operator A() const { A a ; return a;};
    operator C() const { C c ; return c;};
};
void f(A x) {};
int main(){
    B b;
    f((A)b); // The call matches two constructors for A instead of calling operator A()
    return 0;
}
```

Figure 33. Ambiguous user-defined conversions

**Error messages:** Error messages are listed below.

CCN5216: An expression of type B cannot be converted to A.

CCN5219: The call to "A::A" has no best match.

CCN6228: Argument number 1 is an lvalue of type "B".

CCN6202: No candidate is better than "A::A(const A&)".

CCN6231: The conversion from argument number 1 to "const A &" uses the user-defined conversion "B::operator A() const" followed by an lvalue-to-rvalue transformation.

CCN6202: No candidate is better than "A::A(const C &)".

CCN6231: The conversion from argument number 1 to "const C &" uses the user-defined conversion "B::operator C() const".

**Potential solutions:** Possible solutions are listed below.

- Changing `f((A)b)` to the explicit call `f(b.operator A())`
- Removing the constructor `A(const C &)`
- Adding a constructor `A(B)`
- Removing either `operator A()` or `operator C()`

**Note:** The solution you choose depends on your access to classes A and B.

## Issues caused by the use of incomplete types in exception-specifications

A type that is denoted in an exception-specification should not denote an incomplete type. Otherwise, the compiler will diagnose with a severe error where there is an incomplete class type, and an error message is produced. For example:

```
struct MyExcept;
void f1() throw (MyExcept);
```

The compiler is required to produce a diagnostic.

The requirement for a complete class means that templates might be instantiated. For example:

```
template <unsigned N>
struct A {
    __static_assert(N != 0, "Error");
};

void f2() throw (A<0>);
```

The template specialization A<0> is instantiated from the definition of the primary template, resulting in a static assertion error.

## Syntax errors with array new

Prior to z/OS V1R2, C/C++ compilers treated the following two statements as semantically equivalent:

```
new (int *) [1]; /*Syntactially incorrect statement
new int* [1];
```

The first statement is syntactically incorrect even in older versions of the C++ Standard. However, previous versions of C++ accepted it.

As of z/OS V1R2, the C/C++ compiler will produce a compilation error message that specifies the syntactically incorrect statement.



---

## Part 6. Migration issues for C/C++ applications that use other IBM products

The following topics provide information about migration issues resulting from enhancements to the interoperability between XL C/C++ and the other products:

- Chapter 20, "Migration issues with earlier C/C++ applications that run CICS statements," on page 133
- Chapter 21, "Migration issues with earlier C/C++ applications that use DB2 Universal Database," on page 139



---

## Chapter 20. Migration issues with earlier C/C++ applications that run CICS statements

This topic provides information about:

- “Migration of CICS statements from pre-OS/390 C/C++ applications”
- “Migration of CICS statements from earlier XL C/C++ applications” on page 135

---

### Migration of CICS statements from pre-OS/390 C/C++ applications

When you are migrating applications or programs with CICS statements from pre-OS/390 C/C++ applications, be aware of changes and constraints in the following areas:

- “CICS statement translation options”
- “HEAP option used with the interface to CICS”
- “User-developed exit routines”
- “Multiple libraries under CICS”

#### CICS statement translation options

As of z/OS V1R7 XL C/C++ compiler, there is a new option for translating CICS statements into C or C++ code: the z/OS XL C/C++ compiler integrated CICS translator. The standalone CICS translator remains a translation option. For information about when to use the new option, refer to *Translating and compiling for reentrancy in z/OS XL C/C++ Programming Guide, SC14-7315*.

#### HEAP option used with the interface to CICS

In C/370 V2, the location of heap storage under CICS was primarily determined by the residence mode (RMODE) of the program.

With Language Environment services, heap storage is determined only by the HEAP(,ANYWHERE|BELOW) options. RMODE does not affect where the heap is allocated. If the location of heap storage is important, you might want to change the source code accordingly.

#### User-developed exit routines

With Language Environment services in a CICS environment, abnormal termination exit routine CEECDATX is automatically linked at installation time.

This change affects you if you have supplied, or need to supply, your own exit routine. The sample exit routine had been available in the sample library provided with AD/Cycle LE/370 V1R3. It automatically generates a system dump (with abend code 4039) whenever an abnormal termination occurs.

You can modify CEECDATX to suppress the dumps. CEECDATX is available in a z/OS V2R1 XL C/C++ runtime library.

#### Multiple libraries under CICS

You cannot run two different sets of runtime services within one CICS region.

Both the C/370 V2 CICS interface (EDCCICS) and the Language Environment CICS interface could be present in a CICS system through CEDA/PPT definitions and inclusion of modules in the APF STEPLIB. If both interfaces are present, the Language Environment interface will be initialized by CICS when the region is initialized.

You should be aware of changes and constraints in the following areas:

- “CICS abend codes and messages”
- “CICS reason codes”
- “Standard stream support under CICS”
- “Changes in stderr output under CICS” on page 135
- “Transient data queue names under CICS” on page 135

## CICS abend codes and messages

As of z/OS V1R7 XL C/C++ compiler, when you use the CICS option to compile programs with embedded CICS statements, the compiler will issue messages whenever it detects a syntax error before a CICS statement is fully parsed. After a CICS statement is fully parsed, CICS will issue any required messages as described in *CICS Messages and Codes*. The compiler will prepend these CICS messages with product and line numbers and then merge them with the other compiler messages in a single message area.

Abend codes (for example, ACC2) that were used by C/370 V2 under CICS are not issued; the equivalent Language Environment abend code (for example, 4nnn) is issued instead.

### Default option for ABTERMENC changed to ABEND

As of OS/390 V2R9, the default option for ABTERMENC is ABEND instead of RETCODE. If you are expecting the default behavior of ABTERMENC to be RETCODE, you must change the setting in CEECOPT. For details on changing CEECOPT, refer to *z/OS Language Environment Customization, SA38-0685*.

## CICS reason codes

Reason codes that appeared in the CICS message console log have been changed. The current codes are documented in *z/OS Language Environment Debugging Guide*.

## Standard stream support under CICS

With Language Environment services, CICS records sent to the transient data queues associated with stdout and stderr with default settings take the format of the message shown in Figure 34 on page 135.



ASA	terminal id	transaction id	sp	Time Stamp YYYYMMDDHHMMSS	sp	data
1	4	4	1	14	1	108

where:

**ASA** is the carriage-control character

**terminal id**  
is a 4-character terminal identifier

**transaction id**  
is a 4-character transaction identifier

**sp** is a space

**Time Stamp**  
is the date and time displayed in the format YYYYMMDDHHMMSS

**data** is the data sent to the standard streams stdout and stderr.

Figure 34. 1 ASA 4 terminal ID 4 transaction ID 1 sp 14 time stamp 1 sp 108 data

With Language Environment services, CICS records are sent in this format, whether they are directed to the transient data queues associated with stdout and stderr. You should be aware of this change if you are migrating to z/OS V2R1 XL C/C++ compiler, because, previously, this message format had been used for messages directed to the data queue associated with stdout only.

## Changes in stderr output under CICS

Output from stderr is sent to the CICS transient data queue, CESE, which is also used for Language Environment runtime error messages, dumps, and storage reports. If you previously used this file exclusively for C/370 stderr output, you should note that the output might be different than you expect.

## Transient data queue names under CICS

Table 22C/370 transient data queue names are mapped to Language Environment transient data queue names:

Table 22. Transient data queue names under CICS

C/370 name	Language Environment name
CCSI	CESI
CCSO	CESO
CCSE	CESE

## Migration of CICS statements from earlier XL C/C++ applications

When you are migrating applications or programs with CICS statements from earlier C/C++ applications, be aware of the following possibilities:

- "CICS TS V4.1 with "Extended MVS Linkage Convention"" on page 136
- "Customized CEECCSD.COPY and CEECCSDX.COPY files and iconv() changes" on page 136

## CICS TS V4.1 with "Extended MVS Linkage Convention"

The FLOAT(AFP) compiler option instructs the compiler to generate code that uses the full complement of 16 floating-point registers (FPRs). The four original floating-point registers are numbered FPR0, FPR2, FPR4, and FPR6; the additional floating-point (AFP) registers are numbered FPR 1, FPR 3, FPR 5, FPR 7 and FPRs 8 through 15. By convention, FPRs 1, 3, 5, and 7 are always volatile. This means that any called routine could change their values without saving and restoring the original values. However, FPRs 8 through 15 are considered non-volatile by the caller.

In z/OS V1R9 XL C/C++ compiler (and later compilers), FLOAT(AFP) supports the VOLATILE | NOVOLATILE suboption. The default is NOVOLATILE; the compiler assumes that any called subroutines will preserve the values in registers FPRs 8 through 15. It is safe to use NOVOLATILE in most environments, including batch. However, CICS environments prior to CICS TS V4.1 use FPRs 7 through 15 to perform their own task switching. Therefore, you need to specify the FLOAT(AFP(VOLATILE)) option to instruct the compiler to treat FPRs 8 through 15 as volatile.

As of CICS TS V4.1, CICS TS fully supports MVS Linkage conventions. Therefore, if you are compiling floating point code to be run on CICS TS V4.1, you no longer need to use the FLOAT(AFP(VOLATILE)) option.

## Customized CEECCSD.COPY and CEECCSDX.COPY files and iconv() changes

As of z/OS V1R9, load modules for iconv() converters have been renamed in the two CICS sample files CEECCSD.COPY and CEECCSDX.COPY. If your CEECCSD.COPY and CEECCSDX.COPY files have been customized, you need to rename the affected load module entries. Otherwise, the iconv\_open() and iconv\_close() functions cannot distinguish between a customer-created converter and a converter shipped with the Language Environment element.

Language Environment converters are:

- Direct converters (including GENXLT, C and Direct Unicode Converters).
- Indirect Binary converter tables (shipped in <hlq>.SCEEUTBL).
- Indirect Binary converter tables (shipped in the HFS).

### Renaming direct converters

The Direct converters are shipped as load modules in <hlq>.SCEERUN for 31-bit base code, and in <hlq>.SCEERUN2 for XPLINK and 64-bit base code.

**Direct converters for 31-bit base code:** Prior to z/OS V1R9, direct converters for 31-bit base code are shipped as load modules in <hlq>.SCEERUN with a four character prefix of either CEUU or EDCU, with an alias defined for the unshipped prefix. For example, if a given converter's load module has a name of CEUUxxxx, it will also have an alias of EDCUxxxx.

Change the prefix for all 31-bit base direct converters to CEUL. An alias prefix will not be required. In other words:

- A direct converter that was named EDCUxxxx in <hlq>.SCEERUN with an alias of CEUUxxxx will be named CEULxxxx in <hlq>.SCEERUN without an alias.
- A direct converter that was named CEUUxxxx in <hlq>.SCEERUN with an alias of EDCUxxxx will be named CEULxxxx in <hlq>.SCEERUN without an alias.

**Direct converters for XPLINK processing:** Direct converters for XPLINK processing are shipped as load modules in <hlq>.SCEERUN2 with a four character prefix of CEHU. Change the load module prefix for all direct converters for XPLINK to CEHL. In other words, a direct converter that was named CEHUxxxx in <hlq>.SCEERUN2 will be named CEHLxxxx in <hlq>.SCEERUN2.

**Direct converters for 64-bit base code:** Direct converters for 64-bit base code are shipped as load modules in <hlq>.SCEERUN2 with a four character prefix of CEQU. Change the load module prefix for all 64-bit direct converters to CEQL. In other words, a direct converter that was named CEQUxxxx in <hlq>.SCEERUN2 will be named CEQLxxxx in <hlq>.SCEERUN2.

### **Renaming indirect binary converter tables**

Prior to z/OS V1R9, the indirect binary converter tables (ucmap binaries) were shipped in <hlq>.SCEEUTBL with a prefix of EDCU or CEUU, with aliases CEHU for XPLINK and CEQU for 64-bit programs. Change the prefix name for the ucmmap binary converter tables in <hlq>.SCEEUTBL to CEUL, with alias name prefixes of CEHL for XPLINK and CEQL for 64-bit base code. In other words, an indirect binary converter table that was named EDCUxxxx in <hlq>.SCEEUTBL will be named CEULxxxx, with alias names of CEHLxxxx and CEQLxxxx.

### **Renaming HFS indirect binary converter tables**

As of z/OS V1R9, the indirect binary converter tables (ucmap binaries) shipped in the HFS directory /usr/lib/nls/locale/uconvTable are named with a suffix of .libcnvtbl. Add the suffix libcnvtbl to the names of all ucmmap binary converter tables in the HFS directory /usr/lib/nls/locale/uconvTable. In other words, an indirect binary converter table currently named IBM-xxxxx will be renamed to IBM-xxxxx.libcnvtbl.



---

## Chapter 21. Migration issues with earlier C/C++ applications that use DB2 Universal Database

When you are migrating C/C++ applications that use IBM DB2 Universal Database™ services, be aware of the removal of the Database Access Class Library utility.

In addition, beware of the following information:

- “Namespace violations and SQL coprocessor-based compilations”
- “Potential need to specify DBRMLIB with the SQL option” on page 141

**Related information:** See the following related information.

- For more information about the IBM XL C/C++ DB2 coprocessor, refer to “Using the XL C/C++ DB2 coprocessor” in *z/OS XL C/C++ Programming Guide*.
- For detailed information about using these macros with the SQL option, refer to SQL | NOSQL in *z/OS XL C/C++ User’s Guide*.
- For DB2-supplied documentation, see <http://publib.boulder.ibm.com/infocenter/db2luw/v8/>.

---

### Namespace violations and SQL coprocessor-based compilations

As of z/OS V1R10 XL C/C++ compiler, when you use the SQL option for SQL coprocessor-based compilations, you can modify your source code to handle an error condition that would result from using an identifier that has the same name as one of the new predefined but unprotected macros added in this release. The names of unprotected macros are in the preprocessing namespace.

**Note:** Typically, C/C++ compilers treat predefined, unprotected macros as if the source code had been preprocessed with a `#define` directive (such as `#define SQL_VARBINARY_INIT(s) {sizeof(s)-1, s}`).

The XL C/C++ compiler recognizes the following macros as predefined but unprotected:

- `SQL_VARBINARY_INIT`
- `SQL_BLOB_INIT`
- `SQL_CLOB_INIT`
- `SQL_DBCLOB_INIT`

For example, if you use the z/OS V2R1 XL C/C++ compiler to compile the source code shown in Figure 35 with the SQL option, a message will inform you that the macro is already defined.

**Note:** If you use a pre-z/OS V1R10 compiler, you will get undetermined results.

```
--- test.c ---
#define SQL_VARBINARY_INIT 1
--- end test.c ---
```

*Figure 35. Sample source code*

To avoid the error condition you can:

- Perform a macro definition check and handle the error condition, as shown in Figure 36.
- Explicitly undefine the macro, as shown in Figure 38.

## Example: Performing a macro definition check

If you run a macro definition check on the `SQL_..._INIT` identifier, you can specify a preprocessing path that is based on the return code generated by the check.

For example:

- Compiling the code in Figure 36 with the SQL option, and then running it, would generate a return code of "55" if the compiler is z/OS V1R10 XL C/C++ or later, and "66" if a previous version of the compiler is used.
- Compiling the code in Figure 37 with the SQL option, and then running it, would generate a return code of "55".

```
--- test.c ---
#ifdef SQL_VARBINARY_INIT
    int a = 55;
#else
    int a = 66;
#endif

int main(void) {
    return a;
}
--- end test.c ---
```

Figure 36. Portable macro definition check

```
EXEC SQL INCLUDE SQLCA;

int main(void) {
    EXEC SQL BEGIN DECLARE SECTION;
    #ifdef SQL_VARBINARY_INIT
        SQL TYPE IS VARBINARY(100) myvar = SQL_VARBINARY_INIT("abc");
    #else
        SQL TYPE IS VARBINARY(100) myvar = {sizeof("abc")-1, "abc"};
    #endif
    EXEC SQL END DECLARE SECTION;
    return 55;
}
```

Figure 37. Macro definition check and compiler invocation

## Example: Explicitly undefining and redefining a macro

The code in Figure 38 will always be compiled successfully with or without the SQL option because it is completely valid for users to undefine and redefine the various `SQL*_INIT` macros.

```
--- test.c ---
#undef SQL_VARBINARY_INIT
#define SQL_VARBINARY_INIT 1
--- end test.c ---
```

Figure 38. Explicitly undefining a macro

---

## Potential need to specify DBRMLIB with the SQL option

As of z/OS V1R9 XL C/C++ compiler, it is not necessary to specify the DBRMLIB option with the SQL option. For information about using these options, see *z/OS XL C/C++ User's Guide*.

When your source code has embedded SQL statements, you need to use DBRMLIB with SQL only when the specified APARs have been applied to z/OS V1R8 XL C with APAR PK38679.

For more information about using SQL statements, refer to *DB2 Application Programming and SQL Guide*. Useful topics include:

- "Processing SQL statements by using the DB2 coprocessor"
- "Preparing an external SQL procedure by using JCL" (lists the external SQL procedure samples shipped with DB2).

**Note:** The PHASEID compiler option shows the latest PTF that has been applied to the compiler. For detailed information, refer to PHASEID compiler option in *z/OS XL C/C++ User's Guide, SC14-7307*.





---

## Part 7. Appendixes



---

## Appendix. Accessibility

Accessible publications for this product are offered through IBM Knowledge Center (<http://www.ibm.com/support/knowledgecenter/SSLTBW/welcome>).

If you experience difficulty with the accessibility of any z/OS information, send a detailed message to the "Contact us" web page for z/OS (<http://www.ibm.com/systems/z/os/zos/webqs.html>) or use the following mailing address.

IBM Corporation  
Attention: MHVRCFS Reader Comments  
Department H6MA, Building 707  
2455 South Road  
Poughkeepsie, NY 12601-5400  
United States

---

### Accessibility features

Accessibility features help users who have physical disabilities such as restricted mobility or limited vision use software products successfully. The accessibility features in z/OS can help users do the following tasks:

- Run assistive technology such as screen readers and screen magnifier software.
- Operate specific or equivalent features by using the keyboard.
- Customize display attributes such as color, contrast, and font size.

---

### Consult assistive technologies

Assistive technology products such as screen readers function with the user interfaces found in z/OS. Consult the product information for the specific assistive technology product that is used to access z/OS interfaces.

---

### Keyboard navigation of the user interface

You can access z/OS user interfaces with TSO/E or ISPF. The following information describes how to use TSO/E and ISPF, including the use of keyboard shortcuts and function keys (PF keys). Each guide includes the default settings for the PF keys.

- *z/OS TSO/E Primer*
- *z/OS TSO/E User's Guide*
- *z/OS ISPF User's Guide Vol I*

---

### Dotted decimal syntax diagrams

Syntax diagrams are provided in dotted decimal format for users who access IBM Knowledge Center with a screen reader. In dotted decimal format, each syntax element is written on a separate line. If two or more syntax elements are always present together (or always absent together), they can appear on the same line because they are considered a single compound syntax element.

Each line starts with a dotted decimal number; for example, 3 or 3.1 or 3.1.1. To hear these numbers correctly, make sure that the screen reader is set to read out

punctuation. All the syntax elements that have the same dotted decimal number (for example, all the syntax elements that have the number 3.1) are mutually exclusive alternatives. If you hear the lines 3.1 USERID and 3.1 SYSTEMID, your syntax can include either USERID or SYSTEMID, but not both.

The dotted decimal numbering level denotes the level of nesting. For example, if a syntax element with dotted decimal number 3 is followed by a series of syntax elements with dotted decimal number 3.1, all the syntax elements numbered 3.1 are subordinate to the syntax element numbered 3.

Certain words and symbols are used next to the dotted decimal numbers to add information about the syntax elements. Occasionally, these words and symbols might occur at the beginning of the element itself. For ease of identification, if the word or symbol is a part of the syntax element, it is preceded by the backslash (\) character. The \* symbol is placed next to a dotted decimal number to indicate that the syntax element repeats. For example, syntax element \*FILE with dotted decimal number 3 is given the format 3 \\* FILE. Format 3\* FILE indicates that syntax element FILE repeats. Format 3\* \\* FILE indicates that syntax element \* FILE repeats.

Characters such as commas, which are used to separate a string of syntax elements, are shown in the syntax just before the items they separate. These characters can appear on the same line as each item, or on a separate line with the same dotted decimal number as the relevant items. The line can also show another symbol to provide information about the syntax elements. For example, the lines 5.1\*, 5.1 LASTRUN, and 5.1 DELETE mean that if you use more than one of the LASTRUN and DELETE syntax elements, the elements must be separated by a comma. If no separator is given, assume that you use a blank to separate each syntax element.

If a syntax element is preceded by the % symbol, it indicates a reference that is defined elsewhere. The string that follows the % symbol is the name of a syntax fragment rather than a literal. For example, the line 2.1 %OP1 means that you must refer to separate syntax fragment OP1.

The following symbols are used next to the dotted decimal numbers.

**? indicates an optional syntax element**

The question mark (?) symbol indicates an optional syntax element. A dotted decimal number followed by the question mark symbol (?) indicates that all the syntax elements with a corresponding dotted decimal number, and any subordinate syntax elements, are optional. If there is only one syntax element with a dotted decimal number, the ? symbol is displayed on the same line as the syntax element, (for example 5? NOTIFY). If there is more than one syntax element with a dotted decimal number, the ? symbol is displayed on a line by itself, followed by the syntax elements that are optional. For example, if you hear the lines 5 ?, 5 NOTIFY, and 5 UPDATE, you know that the syntax elements NOTIFY and UPDATE are optional. That is, you can choose one or none of them. The ? symbol is equivalent to a bypass line in a railroad diagram.

**! indicates a default syntax element**

The exclamation mark (!) symbol indicates a default syntax element. A dotted decimal number followed by the ! symbol and a syntax element indicate that the syntax element is the default option for all syntax elements that share the same dotted decimal number. Only one of the syntax elements that share the dotted decimal number can specify the ! symbol. For example, if you hear the lines 2? FILE, 2.1! (KEEP), and 2.1 (DELETE), you know that (KEEP) is the

default option for the FILE keyword. In the example, if you include the FILE keyword, but do not specify an option, the default option KEEP is applied. A default option also applies to the next higher dotted decimal number. In this example, if the FILE keyword is omitted, the default FILE(KEEP) is used. However, if you hear the lines 2? FILE, 2.1, 2.1.1! (KEEP), and 2.1.1 (DELETE), the default option KEEP applies only to the next higher dotted decimal number, 2.1 (which does not have an associated keyword), and does not apply to 2? FILE. Nothing is used if the keyword FILE is omitted.

**\* indicates an optional syntax element that is repeatable**

The asterisk or glyph (\*) symbol indicates a syntax element that can be repeated zero or more times. A dotted decimal number followed by the \* symbol indicates that this syntax element can be used zero or more times; that is, it is optional and can be repeated. For example, if you hear the line 5.1\* data area, you know that you can include one data area, more than one data area, or no data area. If you hear the lines 3\* , 3 HOST, 3 STATE, you know that you can include HOST, STATE, both together, or nothing.

**Notes:**

1. If a dotted decimal number has an asterisk (\*) next to it and there is only one item with that dotted decimal number, you can repeat that same item more than once.
2. If a dotted decimal number has an asterisk next to it and several items have that dotted decimal number, you can use more than one item from the list, but you cannot use the items more than once each. In the previous example, you can write HOST STATE, but you cannot write HOST HOST.
3. The \* symbol is equivalent to a loopback line in a railroad syntax diagram.

**+ indicates a syntax element that must be included**

The plus (+) symbol indicates a syntax element that must be included at least once. A dotted decimal number followed by the + symbol indicates that the syntax element must be included one or more times. That is, it must be included at least once and can be repeated. For example, if you hear the line 6.1+ data area, you must include at least one data area. If you hear the lines 2+, 2 HOST, and 2 STATE, you know that you must include HOST, STATE, or both. Similar to the \* symbol, the + symbol can repeat a particular item if it is the only item with that dotted decimal number. The + symbol, like the \* symbol, is equivalent to a loopback line in a railroad syntax diagram.



---

## Notices

This information was developed for products and services offered in the U.S.A. or elsewhere.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing  
IBM Corporation  
North Castle Drive  
Armonk, NY 10504-1785  
U.S.A

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing  
Legal and Intellectual Property Law  
IBM Japan, Ltd.  
19-21, Nihonbashi-Hakozakicho, Chuo-ku  
Tokyo 103-8510, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Site Counsel  
IBM Corporation  
2455 South Road  
Poughkeepsie, NY 12601-5400  
USA

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

#### COPYRIGHT LICENSE:

This information might contain sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

---

## Policy for unsupported hardware

Various z/OS elements, such as DFSMS, HCD, JES2, JES3, and MVS, contain code that supports specific hardware servers or devices. In some cases, this device-related element support remains in the product even after the hardware devices pass their announced End of Service date. z/OS may continue to service element code; however, it will not provide service related to unsupported hardware devices. Software problems related to these devices will not be accepted



for service, and current service activity will cease if a problem is determined to be associated with out-of-support devices. In such cases, fixes will not be issued.

---

## Minimum supported hardware

The minimum supported hardware for z/OS releases identified in z/OS announcements can subsequently change when service for particular servers or devices is withdrawn. Likewise, the levels of other software products supported on a particular release of z/OS are subject to the service support lifecycle of those products. Therefore, z/OS and its product publications (for example, panels, samples, messages, and product documentation) can include references to hardware and software that is no longer supported.

- For information about software support lifecycle, see: IBM Lifecycle Support for z/OS (<http://www.ibm.com/software/support/systemsz/lifecycle/>)
- For information about currently-supported IBM hardware, contact your IBM representative.

---

## Programming interface information

This publication documents *intended* Programming Interfaces that allow the customer to write z/OS XL C/C++ programs.

---

## Standards

The following standards are supported in combination with the Language Environment element:

- The C language is consistent with *Programming languages - C (ISO/IEC 9899:1999)* and a subset of *Programming languages - C (ISO/IEC 9899:2011)*. For more information on ISO, visit their website at <http://www.iso.org>.
- The C++ language is consistent with *Programming languages - C++ (ISO/IEC 14882:1998)*, *Programming languages - C++ (ISO/IEC 14882:2003(E))*, and a subset of *Programming languages - C++ (ISO/IEC 14882:2011)*.

The following standards are supported in combination with the Language Environment and z/OS UNIX System Services elements:

- A subset of *IEEE Std. 1003.1-2001 (Single UNIX Specification, Version 3)*. For more information on IEEE, visit their website at <http://www.iso.org>.
- *IEEE Std 1003.1—1990, IEEE Standard Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C language]*, copyright 1990 by the Institute of Electrical and Electronic Engineers, Inc.
- The core features of *IEEE P1003.1a Draft 6 July 1991, Draft Revision to Information Technology—Portable Operating System Interface (POSIX), Part 1: System Application Program Interface (API) [C Language]*, copyright 1992 by the Institute of Electrical and Electronic Engineers, Inc.
- *IEEE Std 1003.2—1992, IEEE Standard Information Technology—Portable Operating System Interface (POSIX)—Part 2: Shells and Utilities*, copyright 1990 by the Institute of Electrical and Electronic Engineers, Inc.
- The core features of *IEEE Std P1003.4a/D6—1992, IEEE Draft Standard Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API)—Amendment 2: Threads Extension [C language]*, copyright 1990 by the Institute of Electrical and Electronic Engineers, Inc.

- The core features of *IEEE 754-1985 (R1990) IEEE Standard for Binary Floating-Point Arithmetic (ANSI)*, copyright 1985 by the Institute of Electrical and Electronic Engineers, Inc.
- *X/Open CAE Specification, System Interfaces and Headers, Issue 4 Version 2*, copyright 1994 by The Open Group
- *X/Open CAE Specification, Networking Services, Issue 4*, copyright 1994 by The Open Group
- *X/Open Specification Programming Languages, Issue 3, Common Usage C*, copyright 1988, 1989, and 1992 by The Open Group
- United States Government's *Federal Information Processing Standard (FIPS) publication for the programming language C, FIPS-160*, issued by National Institute of Standards and Technology, 1991

---

## Trademarks

IBM, the IBM logo, and `ibm.com` are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at [www.ibm.com/legal/copytrade.shtml](http://www.ibm.com/legal/copytrade.shtml).

Adobe, Acrobat, PostScript and all Adobe-based trademarks are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

---

## Bibliography

This bibliography lists the publications for IBM products that are related to z/OS XL C/C++. It includes publications covering the application programming task. The bibliography is not a comprehensive list of the publications for these products, however, it should be adequate for most z/OS XL C/C++ users. Refer to *z/OS Information Roadmap, SA23-2299*, for a complete list of publications belonging to the z/OS product.

### **z/OS**

- *z/OS Introduction and Release Guide, GA32-0887*
- *z/OS Planning for Installation, GA32-0890*
- *z/OS Summary of Message and Interface Changes, SA23-2300*
- *z/OS Information Roadmap, SA23-2299*
- *z/OS Licensed Program Specifications, GA32-0888*
- *z/OS Migration, GA32-0889*
- *z/OS Program Directory, GI11-9848*

### **z/OS XL C/C++**

- *z/OS XL C/C++ Programming Guide, SC14-7315*
- *z/OS XL C/C++ User's Guide, SC14-7307*
- *z/OS XL C/C++ Language Reference, SC14-7308*
- *z/OS XL C/C++ Messages, GC14-7305*
- *z/OS XL C/C++ Runtime Library Reference, SC14-7314*
- *z/OS C Curses, SA38-0690*
- *z/OS XL C/C++ Compiler and Runtime Migration Guide for the Application Programmer, GC14-7306*
- *Standard C++ Library Reference, SC14-7309*

### **z/OS Metal C Runtime Library**

- *z/OS Metal C Programming Guide and Reference, SC14-7313*

### **z/OS Runtime Library Extensions**

- *z/OS Common Debug Architecture User's Guide, SC14-7310*
- *z/OS Common Debug Architecture Library Reference, SC14-7311*
- *DWARF/ELF Extensions Library Reference, SC14-7312*

### **Debug Tool**

- Debug Tool documentation, which is available at <http://www.ibm.com/software/awdtools/debugtool/library/>.

### **z/OS Language Environment**

- *z/OS Language Environment Concepts Guide, SA38-0687*
- *z/OS Language Environment Customization, SA38-0685*
- *z/OS Language Environment Debugging Guide, GA32-0908*
- *z/OS Language Environment Programming Guide, SA38-0682*
- *z/OS Language Environment Programming Reference, SA38-0683*

- *z/OS Language Environment Runtime Application Migration Guide, GA32-0912*
- *z/OS Language Environment Writing Interlanguage Communication Applications, SA38-0684*
- *z/OS Language Environment Runtime Messages, SA38-0686*

## **Assembler**

- *HLASM Language Reference, SC26-4940*
- *HLASM Programmer's Guide, SC26-4941*

## **COBOL**

- COBOL documentation, which is available at <http://www.ibm.com/software/awdtools/cobol/zos/library/>.

## **PL/I**

- PL/I documentation, which is available at <http://www.ibm.com/software/awdtools/pli/plizos/library/>.

## **VS FORTRAN**

- VS FORTRAN documentation, which is available at <http://www.ibm.com/software/awdtools/fortran/vsfortran/library.html>.

## **CICS Transaction Server for z/OS**

- CICS Transaction Server for z/OS documentation, which is available at: <http://www.ibm.com/software/htp/cics/>

## **DB2**

- DB2 for z/OS documentation, which is available at <http://www.ibm.com/software/data/db2/zos/library.html>.

## **IMS/ESA<sup>®</sup>**

- IMS documentation, which is available at <http://www.ibm.com/software/data/ims/library.html>.

## **MVS**

- *z/OS MVS Program Management: User's Guide and Reference, SA23-1393*
- *z/OS MVS Program Management: Advanced Facilities, SA23-1392*

## **QMF**

- QMF documentation, which is available at <http://www.ibm.com/software/data/qmf/library.html>.

## **DFSMS**

- *z/OS DFSMS Introduction, SC23-6851*
- *z/OS DFSMS Managing Catalogs, SC23-6853*
- *z/OS DFSMS Using Data Sets, SC23-6855*
- *z/OS DFSMS Macro Instructions for Data Sets, SC23-6852*
- *z/OS DFSMS Access Method Services Commands, SC23-6846*

# Index

## Special characters

- `__cplusplus` standard macro
  - determining ISO standard level supported by compiler 115
- `__IBMCPP_EXTENDED_FRIEND` macro
  - as of z/OS V1R11 124
- `__librel()` function 31
  - using to determine library release 31
- `__64` suffix for compiler invocations
  - as of z/OS V1R6 C/C++ 95
- `_CEE_RUNOPTS` environment variable
  - as of z/OS V1R10 105
- `_DEBUG_FORMAT` environment variable 74, 96
  - as of z/OS V1R6 C/C++ 61, 96, 103
    - with LP64 69
- `_EDC_PTHREAD_YIELD` environment variable 107
- `_EDC_PUTENV_COPY` environment variable 82
  - POSIX compliance 73
  - retaining OS/390 behavior 45, 73
  - retaining pre- z/OS V1R5 C/C++ behavior 107
- `_ICONV_MODE` environment variable
  - as of z/OS V1R9 XL C/C++
    - user-defined conversion tables 98
- `_Ieee754.h` header file
  - as of z/OS V1R9 XL C++
    - potential need to include 81
- `_LONG_LONG` macro
  - as of z/OS V1R6 C/C++ 67
- `_OPEN_SYS_SOCKET_IPV6` macro
  - as of z/OS V1R7 XL C++ 82, 100
- `_PVERSION` environment variable
  - as of z/OS V1R8 XL C/C++ 102
- `_TZ` environment variable 47
- `_x` suffix for compiler invocations
  - as of z/OS V1R6 C/C++ 95
- `_XOPEN_SOURCE_EXTENDED` macro
  - as of z/OS V1R9 XL C++ 100
- `@@CTEST` objects
  - relinking C/370 modules 34
- `@euro` locale
  - as of z/OS V1R6 108
- `@preeuro` locale
  - as of z/OS V1R6 108
- `-qplusplus` command option
  - as of z/OS V1R7 XL C/C++
    - when to override 97
- `#pragma` comment
  - and Unicode character translation
    - as of z/OS V1R10 XL C/C++ 63
- `#pragma` enum
  - as of z/OS V1R2 C/C++ 64
- `#pragma` leaves
  - as of OS/390 V2R9 69
- `#pragma` map
  - as of z/OS V1R3 C/C++ 34

- `#pragma` pack(2)
  - as of z/OS V1R2 XL C++
    - unexpected C++ output 84
  - as of z/OS V1R6 C++
    - alignment incompatibilities when binding C and C++
      - modules 102
- `#pragma` reachable
  - as of OS/390 V2R9 69
- `#pragma` runopts
  - pre-OS/390 source code 19
- `#pragma` unroll()
  - as of z/OS V1R7 XL C/C++ 83
- `#pragma` variable
  - as of OS/390 V2R10 C/C++
    - reentrancy 68
  - as of OS/390 V2R9
    - reentrancy 68
  - as of z/OS V1R7 XL C/C++
    - binding OS/390 modules 71

## Numerics

- 32-bit processing
  - as of z/OS V1R6 C/C++
    - default object model 102
- 64-bit processing
  - as of z/OS V1R6 C/C++
    - default object model 102
  - as of z/OS V1R8 XL C/C++
    - GONUMBER compiler option 93
- 64-bit virtual memory
  - as of z/OS V1R8 XL C/C++
    - IPA(LINK) 65
    - IPA(LINK) and ulimit command 99
    - setting MEMLIMIT value 65

## A

- ABEND, compiler
  - as of OS/390 V2R9
    - default option (CICS) 134
  - as of z/OS V1R7 XL C/C++
    - Language Environment codes, under CICS 134
  - as of z/OS V1R8 XL C/C++
    - insufficient storage 65
    - MEMLIMIT system parameter and IMEMLIM variable 65, 99
- abnormal terminations
  - as of OS/390 V2R9
    - Language Environment enclaves 44
  - as of z/OS V1R8 XL C/C++
    - insufficient storage 65
    - changes from C/370 V2 48
    - running pre-OS/390 programs 44

- access-checking
  - as of z/OS V1R2
    - classes (C++ only) 126
- accessibility 145
  - contact IBM 145
  - features 145
- accuracy improvements
  - as of z/OS V1R9
    - IEEE754 math functions 109
- addressing incompatibilities
  - pre-OS/390 20
- AFP registers
  - as of z/OS V1R9
    - CICS processing 136
- alignment incompatibilities
  - as of z/OS V1R6 C/C++
    - between object models 102
  - as of z/OS V1R6 C++
    - binding C and C++ aggregates, both with `#pragma` pack(2) 102
- ambiguous overloads
  - as of z/OS V1R2 C++
    - avoiding 127
- AMODE 64 applications
  - as of z/OS V1R10 105
- ANSI-aliasing rule
  - as of z/OS V1R2 C/C++
    - pointer casting 64
- ANSI/ISO standard compliance
  - freopen() library function 51
- APAR PN74931
  - ILC and pre-OS/390 modules 35
  - pre-OS/390 modules
    - compatibility, achieving 35
- Application Support Class Library from C/C++ for MVS/ESA
  - earlier z/OS C/C++ source code 79
  - OS/390 source code 59
  - pre-OS/390 source code 19
- ARCHITECTURE compiler option
  - as of z/OS V1R2 C/C++ 62
  - as of z/OS V1R6 C/C++
    - and overflow processing 63
  - as of z/OS V2R1 C/C++
    - default 63
- ARGPARSE compiler option
  - as of z/OS V1R13 XL C/C++ 63
- array new
  - as of z/OS V1R2 C/C++
    - avoiding syntax errors 129
  - pre-OS/390 source code
    - with user-defined global new operator 20
- arrays
  - as of V1R9 XL C/C++
    - index definitions 89
- ASA files
  - closing 51
  - closing and reopening 55
  - under CICS 134
  - writing to 51

- assembler interlanguage calls 35
  - pre-OS/390 modules 35
- assembly listings
  - as of z/OS V1R9 XL C/C++
    - width of mnemonic 90
- assembly source
  - System Programming C 22
- assistive technologies 145
- atexit
  - changes from C/370 V2 48

## B

- batch processing
  - as of z/OS V1R2 C/C++
    - alternative 30
    - SYSLIB concatenation 30
  - as of z/OS V1R5
    - abnormal termination exit routine 42
    - CEEBDATX 42
    - CEECDATX 42
  - as of z/OS V1R6 108
  - as of z/OS V1R9 XL C/C++
    - and name mangling 98
  - pre-OS/390 modules
    - abnormal termination exit routines 33
    - CEEBDATX 33
    - CEECDATX 33
    - messages 43
    - MSGFILE runtime option 43
- bibliography 153
- binary compatibility
  - IPA object modules 66
- binder errors
  - as of z/OS V1R8 XL C/C++
    - namespace pollution 33, 101
- binder, invoking
  - as of z/OS V1R8 XL C/C++ 71
- BookManager documents xv
- BPARM proc variable
  - and binder features 102
- as of z/OS V1R8 XL C/C++ 102
- BSD
  - as of z/OS V1R9 XL C++
    - <net/if.h> header file 100
    - socket definitions 100

## C

- C runtime library functions
  - as of OS/390 V2R9
    - pragma requirements 69
- C++ class names
  - as of z/OS V1R3 C/C++ 34
- C++ exception handling
  - as of z/OS V1R2 C++ 122
- C++ Standard compliance
  - 1998 support 69
  - as of z/OS V1R7 XL C/C++ 80
- c++ utility
  - as of z/OS V1R6 C/C++
    - g flag translation 29

- C++11
  - as of z/OS V1R11
    - WARN0X compiler option 126
- C++11 compiler option
  - as of z/OS V2R1 122
- c89 utility 95
  - g flag option 103
  - as of z/OS V1R6 C/C++ 96, 103
    - g flag option 74
    - g flag translation 29
  - binding OS/390 modules 74
  - debug format 61
  - as of z/OS V1R8 XL C/C++ 71, 102
  - debug format
    - as of z/OS V1R6 C/C++ 96
  - feature specification
    - as of z/OS V1R8 XL C/C++ 102
- C99 support
  - as of z/OS V1R7 XL C++
    - standard macros 83
    - TARGET compiler option 83
  - as of z/OS V1R9
    - IEEE754 math functions 109
    - runtime libraries 109
  - hexadecimal floating point notation 110
  - numeric conversion functions 111
- catalogued procedures
  - and binder features 102
  - as of z/OS V1R8 XL C/C++ 102
    - IMEMLM variable 65
    - IPA Link 65
- CBCI procedure
  - as of x/OS V1R5 C++
    - compiling OS/390 applications 70
  - pre-z/OS V1R5 programs 99
- CBCXI procedure
  - as of x/OS V1R5 C++
    - compiling OS/390 applications 70
  - pre-z/OS V1R5 programs 99
- CC command
  - syntax, supporting old, new, or both 70
- CC EXEC
  - as of V1R2
    - invocation syntax changes 33
- CC EXEC statement
  - customization of 70
- CCN5193 exception
  - as of z/OS V1R2
    - avoiding 127
- CCN5413 exception
  - as of z/OS V1R9
    - avoiding 126
- CEEBDATX procedure
  - as of z/OS V1R5 42
  - pre-OS/390 modules 33
- CEEBINT High-Level Language exit routine
  - with setenv() function call 73, 106
- CEEBINT High-Level Language exit routines
  - with setenv() function call 41
- CEEBLIBIA library module 46
  - environment initialization 47
- CEEBXITA library module
  - rules of precedence 33

- CEECDATX procedure 133
  - pre-OS/390 modules 33
- CEECOPT procedure
  - under CICS
    - as of OS/390 V2R9 134
- CEEDOPT procedure
  - as of OS/390 V2R9
    - abnormal terminations of enclaves 44
- CEESTART library module 46
  - initialization compatibility 46
- CHECKOUT compiler option
  - as of z/OS V1R13 XL C/C++ 91
  - as of z/OS V1R6 C/C++
    - C support 64
- CHECKOUT(CAST) compiler option
  - as of z/OS V1R2 C/C++ 64
- CICS
  - abend codes and messages 134
  - API 135
  - heap residence 133
  - reason codes 134
  - standard stream support 134
  - stderr 135
  - transient data queue names 135
  - using HEAP option 133
- CICS processing
  - as of z/OS V1R9
    - AFP registers 136
    - FLOAT(AFP) compiler option 136
    - iconv() changes and CEECCSD.COPY and CEECCSDX.COPY files 136
    - Load Module Analyzer (LMA) 136
- CICS statement translation options
  - as of z/OS V1R7 133
- class definitions
  - as of z/OS V1R2
    - avoiding exceptions 127
    - CCN5193 exception 127
    - type definitions 127
  - as of z/OS V1R9
    - CCN5413 exception 126
    - class access checking 126
- class libraries
  - changes between z/OS V1R5 C/C++ and z/OS V2R1 XL C/C++
    - no longer supported 79
  - class library incompatibilities
    - earlier z/OS C/C++ source code 79
  - IO Stream Class
    - earlier z/OS C/C++ source code 79
    - load module 75
    - OS/390 source code 59
    - pre-OS/390 source code 19
    - source code 75
    - OS/390 source code 59
    - pre-OS/390 source code 19
- CLBPRFX variable
  - as of x/OS V1R5 C++
    - compiling OS/390 applications 70
  - pre-z/OS V1R5 programs 99
- CLISTS
  - changes affecting pre-OS/390 programs 43

CMDOPTS compiler option  
   as of z/OS V1R7 XL C/C++ 91

COBOL interlanguage calls  
   pre-OS/390 modules 35, 36

code points  
   no longer supported  
   pre-OS/390 source code 23

Collection Class Library from C/C++ for MVS/ESA  
   earlier z/OS C/C++ source code 79  
   OS/390 source code 59  
   pre-OS/390 source code 19

command-line parameters  
   Language Environment error handling 42  
   passing to a program 42

comments, using  
   as of z/OS V1R7 XL C/C++  
   when to override `-qcpluscmt` 97

Communications Server information handling  
   as of z/OS V1R9 XL C/C++ 81

COMPAT binder option  
   and c89 utility 102  
   as of z/OS V1R8 XL C/C++ 102

COMPAT compiler option  
   as of z/OS V1R6 C/C++ 96

compat object model  
   as of z/OS V1R6 C/C++ 102

compatibility issues  
   bind-time  
     from pre-OS/390 to z/OS V1R9 31  
     OS/390 71  
   C/370 Common Library  
     as of z/OS V1R9 46  
   compile-time  
     earlier z/OS C/C++ programs 87  
   I/O operations  
     from pre-OS/390 51  
   initialization sequence  
     interception 46  
   input and output  
     from pre-OS/390 51  
   IPA release-to-release binary compatibility 66  
   runtime  
     OS/390 applications 73  
     pre-OS/390 applications 45  
   source code  
     earlier z/OS C/C++ programs 79  
     OS/390 programs 59  
     pre-OS/390 compiler to z/OS V1R9 XL C/C++ 19  
   compatibility, achieving  
     pre-OS/390 modules  
       APAR PN74931 35  
       upward and downward 35  
       with earlier and later releases 35  
     with earlier and later releases  
       compatibility, achieving 35  
   compile-time issues  
     from pre-OS/390 25  
   compiler invocations  
     as of z/OS V1R6 C/C++ 95  
     c89 61, 96

compiler messages, listings, and return codes  
   ongoing changes and dependencies 25, 61, 87

compiler option  
   LP64 95  
   TARGET  
     as of z/OS V1R12 XL C/C++ 95  
   XPLINK 95

compiler options  
   ARCHITECTURE  
     as of z/OS V1R2 C/C++ 62  
     as of z/OS V2R1 C/C++ 63  
   CHECKOUT  
     C support as of z/OS V1R6 C/C++ 64  
   CHECKOUT(CAST)  
     as of z/OS V1R2 C/C++ 64  
   COMPAT  
     as of z/OS V1R6 C/C++ 96  
   DBRMLIB  
     as of z/OS V1R8 XL C 141  
     z/OS V1R5 XL C — z/OS V1R8 XL C 141  
   DECK 25  
     alternative as of z/OS V1R2 C/C++ 62  
   DIGRAPH  
     default as of z/OS V1R2 C/C++ 64  
   ENUM  
     as of z/OS V1R2 C/C++ 64  
   ENUMSIZE  
     as of z/OS V1R2 C/C++ 26, 64  
     as of z/OS V1R7 XL C/C++ 26  
   ENUMSIZE(SMALL)  
     as of z/OS V1R7 XL C++ 92  
   FLAG 93  
   GENPCH  
     as of z/OS V1R2 C/C++ 62  
   GONUMBER  
     with LP64 93  
   HALT 26  
   HALTONMSG  
     as of z/OS V1R2 C/C++ 64  
   HWOPTS  
     alternative as of z/OS V1R2 C/C++ 62  
     as of z/OS V1R2 C/C++ 26  
   ILP32  
     as of z/OS V1R9 XL C/C++ 98  
     batch processing and name mangling under ILP32 98  
   INFO  
     C support as of z/OS V1R6 27, 64  
     C support as of z/OS V1R6 C/C++ 64  
   INLINE  
     as of z/OS V1R2 C/C++ 27, 64  
   IPA 66  
     as of z/OS V1R8 XL C 29, 70, 98  
     as of z/OS V2R1 XL C/C++ 93  
   LANGLVL  
     as of z/OS V1R7 XL C/C++ 93, 94

compiler options (*continued*)  
   LANGLVL(ANSI) compiler option  
     as of z/OS V1R7 XL C 25, 27, 62, 66, 90, 93  
   LANGLVL(COMPAT) 26  
     as of z/OS V1R2 C/C++ 62  
   LANGLVL(EXTENDED) compiler option  
     as of z/OS V1R7 XL C 27, 66, 94  
   LANGLVL(EXTERNTEMPLATE) compiler option  
     as of z/OS V1R11 125  
   LANGLVL(SAA) compiler option  
     as of z/OS V1R7 XL C 25, 27, 62, 66, 90, 93  
   LANGLVL(SAA2) compiler option  
     as of z/OS V1R7 XL C 25, 27, 62, 66, 90, 93  
   LOCALE  
     as of z/OS V1R9 67  
     as of z/OS V1R9 XL C/C++ 94  
   LSEARCH 28, 62  
     as of z/OS V1R2 C/C++ 62  
   NORENT  
     as of OS/390 V2R9 68  
     as of z/OS V1R7 XL C/C++ 71  
   OE  
     as of z/OS V1R2 C/C++ 62  
   OMVS 26  
     alternative as of z/OS V1R2 C/C++ 62  
   OPTIMIZE 28  
     as of z/OS V1R5 C/C++ 68  
   ROCONST  
     default as of z/OS V1R2 C/C++ 68  
   ROSTRING  
     as of z/OS V1R2 C/C++ 68  
   SEARCH 28  
     as of z/OS V1R2 C/C++ 62  
   SOM  
     as of OS/390 V2R10 C/C++ 62  
     no longer supported 62  
   SQL  
     as of z/OS V1R8 XL C 141  
   SRCMSG 26  
     as of z/OS V1R2 C/C++ 62  
   STATICINLINE  
     default as of z/OS V1R2 C/C++ 68  
   SYSLIB 26  
     alternative as of z/OS V1R2 C/C++ 62  
   SYSPATH 26  
     alternative as of z/OS V1R2 C/C++ 62  
   TARGET  
     as of z/OS V1R6 C/C++ 96  
     as of z/OS V1R9 XL C/C++ 69  
   TEST  
     as of z/OS V1R6 C/C++ 29, 69  
   TUNE  
     as of z/OS V2R1 C/C++ 69  
   USEPCH  
     as of z/OS V1R2 C/C++ 62  
   USERLIB 26

- compiler options (*continued*)
  - alternative as of z/OS V1R2 C/C++ 62
  - USERPATH 26
  - alternative as of z/OS V1R2 C/C++ 62
- compiler options for compatibility with previous compilers 99
- compiler options, no longer supported as of z/OS V1R2 C/C++ 62
- compiler options, specifying in JCL 29
- compiler options, no longer supported pre-OS/390 25
- compiler substitution variables as of z/OS V1R10 88
- compiler-time issues from C/370 V2 25
- concatenation of libraries
  - environment initialization 47
- conflicts between options and pragmas as of z/OS V1R7 XL C/C++ 91
- contact
  - z/OS 145
- ctest() function
  - relinking C/370 modules 34
- ctime() 47
- customization
  - as of z/OS V1R6
    - Language Environment services 108
- cv-qualification
  - as of z/OS V1R2 C++ 122

## D

- data conversions
  - as of z/OS V1R6 C/C++ and ARCHITECTURE level 63
- data set names 69
- data type incompatibilities pre-OS/390 source code 21
- data types
  - as of z/OS V1R6 XL C long long 67
- Database Access Class Library
  - as of OS/390 V1R4
    - removal of utility 75
- DB2 services, requesting
  - using SQL compiler option 139
- DB2 Universal Database
  - Database Access Class Library utility 139
  - requesting DB2 services
    - z/OS V1R5 XL C — z/OS V1R8 XL C 139
- DBRMLIB compiler option
  - z/OS V1R5 XL C — z/OS V1R8 XL C 141
- ddnames
  - SYSERR 43
  - SYSPRINT 43
  - SYSTEM 43
- debug format 103
  - as of z/OS V1R6 C/C++
    - binding OS/390 modules 74
    - c89 utility 61
    - determining 69

- debug format (*continued*)
  - c89 utility 96
- Debug Tool
  - relinking C/370 modules 34
- debugging issues
  - relinking C/370 modules 34
- decimal floating-point (DFP)
  - as of z/OS V1R9 XL C++
    - size modifiers 82, 106
- decimal overflow exceptions
  - pre-OS/390 CICS modules 48
- DECK compiler option
  - alternative as of z/OS V1R2 C/C++ 25
  - as of z/OS V1R2 C/C++
    - alternative 62
- default daylight saving time
  - as of z/OS V1R9 107, 108
- destruction of statically initialized objects
  - before and after ISO/IEC 14882:2003(E) compliance 120
- DIGRAPH compiler option
  - as of z/OS V1R2 C/C++
    - default 64
- DSAUSER compiler option
  - as of z/OS V1R13 XL C/C++ 92
- DSECT header files
  - packed structures and unions 21
- dump services
  - as of C/C++ for MVS/ESA V3
    - dump generation or suppression 43
- dumps
  - generating automatically
    - as of z/OS V1R5 42
  - Language Environment format
    - as of z/OS V1R5 42
- DWARF debug format
  - g flag
    - as of z/OS V1R6 C/C++ 96
- dynamic binding
  - declaring and calling virtual functions
    - as of z/OS V1R6 C/C++ 84
- dynamic code 46

## E

- EDCXSTRX
  - and dynamic C library functions in SPC applications 22
- EDCXV 22
- EEC default currency
  - as of z/OS V1R6 108
- enclaves
  - as of OS/390 V2R9
    - abnormal terminations 44
- enumeration types
  - as of z/OS V1R7 XL C/C++
    - controlling size of 64
  - controlling size of
    - as of z/OS V1R7 XL C/C++ 26
    - as of z/OS V1R7 XL C++ 92
- enumerations
  - as of z/OS V1R7 XL C++ 92
- differences between UNIX System Laboratories and Standard C++ I/O Stream libraries 75

- ENUMSIZE compiler option
  - as of z/OS V1R2 C/C++ 26, 64
  - as of z/OS V1R7 XL C/C++ 26
- ENUMSIZE(SMALL) compiler option
  - as of z/OS V1R7 XL C++ 92
- ENVAR("\_EDC\_COMPAT=32767")
  - runtime option 41, 73, 106
- environment initialization 47
- environment variables
  - \_EDC\_COMPAT 53
  - as of z/OS V1R5 C/C++
    - POSIX compliance 82
    - putenv() 82
    - storage of 82
  - as of z/OS V1R6 C/C++
    - \_DEBUG\_FORMAT 29, 61
    - g flag translation 29, 61
    - c89/c++ 29
    - DWARF 61
  - internationalization issues 47
  - POSIX compliance 47
- error messages
  - as of z/OS V1R8 XL C/C++
    - binder 33, 101
  - as of z/OS V1R9 XL C++
    - name lookup exceptions 90
    - templates 90
  - Language Environment services
    - redirecting 43
  - namespace pollution
    - as of z/OS V1R8 XL C/C++ 33, 101
  - templates 90
- errors
  - as of z/OS V1R7 XL C++
    - non-standard long long macros 83
  - due to compiler changes 126
- errors, migration
  - macro redefinitions
    - as of z/OS V1R7 XL C 25, 27, 62, 66, 90, 93
  - Unable to open DBRM file
    - as of z/OS V1R8 XL C 141
- escape sequence encoding
  - as of z/OS V1R11 88
- Euro
  - as of z/OS V1R6 108
- exception handling
  - as of z/OS V1R2
    - access checking (C++ only) 126
    - class type definitions 126
  - as of z/OS V1R2 C++ 122
    - ambiguous overloads 127
  - as of z/OS V1R9
    - CCN5413 exception 126
- changes from C/370 V2
  - return codes 48
  - SIGINT 48
  - SIGTERM 48
  - SIGUSR1 48
  - SIGUSR2 48
- differences between C/370 and Language Environment
  - library return codes and messages 41
  - user-defined conversions 128



- exceptions
  - as of z/OS V1R2
    - avoiding exceptions 127
    - CCN5193 exception 127
    - type definitions 127
- EXEC statements
  - CC 33
  - CC command 70
  - changes affecting pre-OS/390 programs 43
  - customization of 70
- existing applications, migrating to z/OS XL C
  - From C/370 V2 17
- external references
  - as of z/OS V1R3 C/C++ 34
- external variable names
  - as of z/OS V1R3 C/C++ 34

**F**

- feature test macros
  - and system header files
    - as of z/OS V1R9 XL C++ 81
- feature testing
  - as of z/OS V1R11 XL C++ 100
  - as of z/OS V1R7 XL C++ 82, 100
  - as of z/OS V1R9 XL C++ 100
- fetchd main programs
  - pre-OS/390 source code 21
- fflush() function 53
- fgetpos() function 53
- fixes
  - pre-OS/390 modules
    - APAR PN74931 35
  - z/OS V1R5 XL C — z/OS V1R8 XL C
    - DBRMLIB option 139
- FLAG compiler option
  - as of z/OS V1R13 XL C/C++ 93
- flags
  - differences between UNIX System Laboratories and Standard C++ I/O Stream libraries 75
- fldata() function
  - changes in return values 55
- FLOAT(AFP) compiler option
  - CICS processing
    - as of z/OS V1R9 136
- floating-point support
  - runtime libraries 110
- for loops
  - as of z/OS V1R7 XL C/C++
    - unrolling 83
  - scoping
    - as of z/OS V1R2 C++ 120
- format control flags
  - differences between UNIX System Laboratories and Standard C++ I/O Stream libraries 75
- Fortran interlanguage calls
  - as of Language Environment V1R5 35
- freopen() library function
  - ANSI/ISO standard 51
- friend declaration
  - as of z/OS V1R11
    - extendedfriend 124

- friend declarations in class member lists
  - and Standard C++ compliance
    - as of z/OS V1R2 C++ 121
- friend declarations, visibility of
  - as of z/OS V1R2 C++
    - effect on friend declarations 121
- fseek() function 53
- function return type
  - pre-OS/390 source code 20

**G**

- GENASM compiler option
  - as of z/OS V1R13 XL C/C++ 93
- GENPCH compiler option
  - as of z/OS V1R2 C/C++ 62
- getnameinfo() function
  - as of z/OS V1R9 XL C/C++
    - scope information 81
- global new operator, user-defined
  - pre-OS/390 source code
    - example 20
- GONUMBER compiler option
  - as of z/OS V1R8 XL C/C++
    - with LP64 93

**H**

- HALT compiler option 26
- HALTONMSG compiler option
  - as of z/OS V1R2 C/C++ 64
- header files
  - and feature test macros
    - as of z/OS V1R9 XL C++ 81
  - as of z/OS V1R7 XL C++
    - \_OPEN\_SYS SOCK\_IPV6 macro 82
    - exposing new definitions 82
    - Language Environment 82, 100
  - as of z/OS V1R9
    - time.h 107
  - as of z/OS V1R9 XL C++ 81
    - \_Ieee754.h 81
    - IEEE 754 interface declarations 81
    - Language Environment 100
  - DSECT
    - migration from pre-OS/390 21
- HEAP runtime option
  - default size 44
  - parameters 44
  - with CICS 133
- HEAPOOLS runtime option
  - as of z/OS V1R10 105
- hexadecimal floating point notation
  - C99 support 110
- HFS files, support of 70
- HWOPTS compiler option
  - as of z/OS V1R2 C/C++ 26
    - alternative 62

**I**

- IBM data set names 69
- IBM object model
  - as of z/OS V1R6 C/C++ 102

- IBM Open Class Library
  - OS/390 source code 59
  - earlier z/OS C/C++ source code 79
  - pre-OS/390 source code 19
  - removal of runtime support 75
- IBMBLIIA library module 46
  - environment initialization 47
- IBMBXITA library module
  - rules of precedence 33
- iconv() changes and CICS processing
  - as of z/OS V1R9 136
- IEEE 754 interface declarations
  - as of z/OS V1R9 XL C++
    - namespace pollution 81
- IEEE754 math functions
  - as of z/OS V1R9
    - version specification 109
- IEFUSI exit routine
  - as of z/OS V1R8 XL C/C++
    - MEMLIMIT value 99
    - MEMLIMITvalue 65
- IEW2456E error condition
  - binding earlier z/OS C/C++ programs
    - handling 101
  - binding pre-OS/390 programs
    - handling 33
- ILP32 compiler option
  - as of z/OS V1R9 XL C/C++
    - batch processing and name mangling 98
- IMEMLIM variable
  - as of z/OS V1R8 XL C/C++
    - cataloged procedures 99
    - MEMLIMIT system parameter 99
    - to override the MEMLIMIT default 65
- implicit integer types
  - as of z/OS V1R2 C++ 120
- include files, finding 26
- incompatibilities
  - between Open Class and Standard /C++ libraries 75
- INFO compiler option
  - as of z/OS V1R6 C/C++
    - C support 64
    - C support as of z/OS V1R6 27
    - default as of z/OS V1R2 C/C++ 27
- initialization compatibility issues 46
  - C/370 Common Library
    - as of z/OS V1R9 46
- initialization schemes
  - CEESTART and IBMBLIIA modules 46
- INLINE compiler option
  - as of z/OS V1R2 C/C++ 27
    - defaults 64
- inlining threshold
  - as of z/OS V1R2 C/C++ 64
- input and output
  - as of z/OS V1R9 XL C++
    - impact of DFP size modifiers 82
    - impact of DFP size modifiers on fprintf/fscanf results 106
    - source code modifications to fprintf and fscanf function arguments 82

- input and output (*continued*)
    - ASA files
      - closing and reopening 55
      - closing files 51
      - writing to files 51
    - closing and reopening files
      - ASA files, opening and closing 55
    - closing files
      - ASA files 51
    - compatibility issues 51
    - file I/O changes 51
    - fldata() function 55
    - ftell() encoding 53
    - opening files 51
    - repositioning within files 53
    - terminal I/O 56
    - VSAM I/O 55
    - writing to files
      - ASA files 51
      - other considerations 51
  - interlanguage calls
    - assembler 35
    - PL/I 35
  - interlanguage calls (ILC)
    - as of Language Environment V1R5 35
    - as of z/OS V1R6 C++
      - between C and C++ program modules using #pragma pack(2) 102
    - pre-OS/390 binder error 48
    - pre-OS/390 modules 35
    - pre-OS/390 source code 22
    - program mask manipulations
      - pre-OS/390 source code 22
    - relinking pre-OS/390 modules 34
  - internal timing algorithm
    - as of z/OS V1R8 107
  - internationalization
    - migration issues 108
  - internationalization incompatibilities
    - no longer supported
      - pre-OS/390 source code 23
    - pre-OS/390 source code 23
  - internationalization issues
    - time zones 47
  - invocation of XL C/C++ compiler
    - as of z/OS V1R6 C/C++ 95
  - IPA compiler option
    - as of z/OS V1R9 XL C/C++
      - IPA link step 70
      - macro redefinition 70
      - region size 70
      - very large applications 70
    - as of z/OS V2R1 XL C/C++ 93
    - binary compatibility issues 66
    - macro redefinition
      - as of z/OS V1R8 XL C 29, 98
  - IPA Link step
    - as of z/OS V1R9 XL C/C++
      - very large applications 70
    - very large applications
      - as of z/OS V1R8 XL C 29, 98
  - IPA(LINK) compiler option
    - as of z/OS V1R8 XL C/C++ 65
    - 64-bit memory 99
    - link step defaults 65
  - ISAINC runtime option
    - Language Environment equivalent 43
  - ISASIZE runtime option
    - Language Environment equivalent 43
  - ISASIZE/ISAINC with #pragma runopts
    - pre-OS/390 source code 19
  - ISO standard C++ compliance
    - determining level supported by compiler 115
  - ISO Standard C++ compliance 99
    - recommended approaches for migration objectives 117
  - ISO/IEC 14882:2003(E) compliance
    - effect on cv-qualification 122
    - statically initialized objects, destruction of 120
- J**
- JCL procedures
    - arguments that contain a slash 43
    - as of C/C++ for MVS/ESA V3
      - dump generation or suppression 43
    - as of x/OS V1R5 C++
      - CBCI 70
      - CBXI 70
      - CLBPRFX variable 70
    - as of z/OS V1R2 C/C++ 30
    - as of z/OS V1R5
      - CEEBDATX 42
    - as of z/OS V1R5 C/C++ 30
    - as of z/OS V1R7 XL C/C++
      - bind step 71
    - as of z/OS V1R8 XL C/C++
      - 64-bit virtual memory 65
      - setting MEMLIMIT value 65
    - as of z/OS V1R9 XL C/C++
      - default region size 97
      - name mangling 98
      - user-defined conversion tables 98
  - CBCC 29
  - CBCCL 29
  - CBCCLG 29
  - CBCI 30
  - CBCXI 30
  - CC EXEC statement 70
  - CEEBDATX 33
  - CEECDATX 33, 133
  - CEECOPT
    - as of OS/390 V2R9 134
  - CEEDOPT
    - abnormal terminations of enclaves 44
  - changes affecting pre-OS/390 programs 43
  - CLBPRFX variable 30
  - customizing for migrations from OS/390 70
  - CXX parameter 29
  - differences between C/370 and AD/Cycle C/370 V1R2
    - library return codes and messages 41
  - GO step 43
- K**
- keyboard
    - navigation 145
    - PF keys 145
    - shortcut keys 145
- L**
- LANGLVL compiler option
    - and macro redefinitions
      - as of z/OS V1R7 XL C/C++ 93, 94
  - LANGLVL(ANSI) compiler option
    - and Standard C++ compliance objectives 117
    - as of z/OS V1R7 XL C
      - macro redefinition 62, 66
    - macro redefinition
      - as of z/OS V1R7 XL C 25, 27, 90, 93
  - LANGLVL(AUTOTYPEDEDUCTION) compiler option
    - as of z/OS V1R12 123
  - LANGLVL(C1XNORETURN) compiler option
    - as of z/OS V2R1 123
  - LANGLVL(C99LONGLONG) compiler option
    - as of z/OS V1R12 123
  - LANGLVL(C99PREPROCESSOR) compiler option
    - as of z/OS V1R12 123
  - LANGLVL(COMPAT) compiler option
    - alternative as of z/OS V1R2 C/C++ 26
    - as of z/OS V1R2 C/C++ 62
  - LANGLVL(COMPAT92) compiler option
    - and Standard C++ compliance objectives 117
  - LANGLVL(CONSTEXPR) compiler option
    - as of z/OS V2R1 124
  - LANGLVL(DECLTYPE) compiler option
    - as of z/OS V1R12 124
  - LANGLVL(DEFAULTANDDELETE) compiler option
    - as of z/OS V2R1 124
  - LANGLVL(DELEGATINGCTORS) compiler option
    - as of z/OS V1R12 124
  - LANGLVL(EXPLICITCONVERSIONOPERATORS) compiler option
    - as of z/OS V2R1 124
- JCL procedures (*continued*)**
- interlanguage calls and compiler options 36
  - obsolete C/370 runtime options 43
  - pre-z/OS V1R5 C/C++ modifications 99
  - SYSLIB DD cards to remove
    - as of z/OS V1R2 C/C++ 30
  - to compile very large applications
    - as of z/OS V1R8 XL C 29, 70, 98
  - user-defined for C++ 29

- LANGLVL(EXTC1X) compiler option
  - as of z/OS V2R1 93
- LANGLVL(EXTENDED) compiler option
  - and Standard C++ compliance objectives 117
  - as of z/OS V1R7 XL C
    - macro redefinition 66
    - macro redefinition as of z/OS V1R7 XL C 27, 94
- LANGLVL(EXTENDED0X) compiler option
  - as of z/OS V1R11 94
- LANGLVL(EXTENDEDFRIEND) compiler option 124
- LANGLVL(EXTENDEDINTEGERSAFE) compiler option
  - as of z/OS V1R12 125
- LANGLVL(EXTENTTEMPLATE) compiler option
  - as of z/OS V1R11
    - macro redefinition 125
- LANGLVL(IMPLICITINT) compiler option 120
- LANGLVL(INLINENAMESPACE) compiler option
  - as of z/OS V1R12 125
- LANGLVL(LONGLONG) compiler option
  - as of z/OS V1R7 XL C++ 83
- LANGLVL(NOANSIFOR) compiler option
  - scoping for-loop initializer declarations
    - as of z/OS V1R2 C++ 120
- LANGLVL(OLDFRIEND) compiler option
  - as of z/OS V1R2 C++
    - effect on friend declarations 121
- LANGLVL(OLDMATH) compiler option
  - as of z/OS V1R2 C++ 127
- LANGLVL(REFERENCECOLLAPSING) compiler option
  - as of z/OS V2R1 125
- LANGLVL(RIGHTANGLEBRACKET) compiler option
  - as of z/OS V2R1 125
- LANGLVL(RVALUEREFERENCES) compiler option
  - as of z/OS V2R1 126
- LANGLVL(SAA) compiler option
  - as of z/OS V1R7 XL C
    - macro redefinition 62, 66
    - macro redefinition as of z/OS V1R7 XL C 25, 27, 90, 93
- LANGLVL(SAA2) compiler option
  - as of z/OS V1R7 XL C
    - macro redefinition 62, 66
    - macro redefinition as of z/OS V1R7 XL C 25, 27, 90, 93
- LANGLVL(SCOPEDENUM) compiler option
  - as of z/OS V2R1 126
- LANGLVL(STATIC\_ASSERT) compiler option
  - as of z/OS V1R12 126
- LANGLVL(STRICT98) compiler option
  - and Standard C++ compliance objectives 117
- LANGLVL(VARIADICTEMPLATES) compiler option
  - as of z/OS V1R12 126
- Language Environment
  - as of z/OS V1R7 XL C++ 82
  - header files
    - as of z/OS V1R7 XL C++ 82
    - netinet/in.h 82
- Language Environment runtime libraries
  - as of z/OS V1R7 XL C++
    - header files 100
    - header files 100
  - pre-OS/390 modules
    - packaging 36
- Language Environment services
  - as of OS/390 V2R9
    - abnormal enclave terminations 44
    - abnormal terminations 44
    - enclaves 44
  - as of z/OS V1R2 C/C++
    - arguments that contain a slash 43
    - data set names 43
    - default heap allocatons 44
    - error messages 41, 42
    - error parameter passing 42
    - HEAP parameter specification 44
    - passing runtime options 43
    - return codes 41
    - STACK defaults 45
    - TRAP restrictions 44
  - as of z/OS V1R5 C/C++
    - abnormal terminations 42
    - batch jobs 42
    - customizing procedures 99
    - data set names 42
    - modifying JCL 99
    - specifying message language 42
  - as of z/OS V1R6
    - customization 108
    - LOCALDEF utilities 108
  - as of z/OS V1R7 XL C/C++
    - abend codes and messages with CICS 134
    - dumps 134
  - as of z/OS V1R9
    - default daylight saving time 107
    - default daylight saving time, retaining previous 108
  - C/370 CICS modules
    - initialization compatibility issues 46
    - realloc() 49
    - unexpected SIGFPE exceptions 48
  - CICS modules
    - writing to pre-OS/390 files 51
  - customization issues
    - OS/390 migrations 74
  - equivalents for C/370 V2 runtime options 43
  - iconv() changes and CICS processing
    - as of z/OS V1R9 136
  - initialization 46
  - interlanguage calls (ILC) 35
- Language Environment services
  - (continued)
  - OS/390 migration issues
    - customization 74
  - output handling under CICS 135
  - pre-OS/390 CICS modules
    - coexistence considerations 48
    - decimal overflow exceptions 48
    - exception handling 48
    - initialization schemes 46
    - initializing 47
    - input and output compatibility issues 51
  - pre-OS/390 CICS programs
    - abnormal terminations 133
    - dumps 133
    - heap residence 133
  - pre-OS/390 modules
    - APAR PN74931 35
    - converting modules to use Language Environment services 38
    - directing error messages 43
  - pre-OS/390 programs
    - retaining runtime behavior 41
    - runtime messages 41
    - STACK parameters 45
  - record handling under CICS 134
  - transient data queue names under CICS 135
- language for compiler messages, specifying 61
- language libraries
  - pre-OS/390 modules 36
- LANGUAGE runtime option
  - Language Environment equivalent 43
- LANGUAGE with #pragma runopts
  - pre-OS/390 source code 19
- LC\_MONETARY information
  - as of z/OS V1R6 108
- library file searches
  - based on name and type
    - as of z/OS V1R2 C/C++ 30
- library functions
  - ctest() 34
  - ctime() 47
  - fflush() 53
  - fgetpos() 53
  - fseek() 53
  - librel 31
  - localtime() 47
  - mktime() 47
  - pthread\_yield()
    - as of z/OS V1R8 XL C/C++ 80
  - pthread\_yield() function
    - as of z/OS V1R9 XL C++ 80
  - putenv()
    - as of z/OS V1R5 C/C++ 82, 107
  - realloc()
    - migration from pre-OS/390 49
    - pre-OS/390 source code modification 49
  - sched\_yield()
    - as of z/OS V1R8 XL C/C++ 80
  - unsetc() 53

- library release
  - determining 31
- link step
  - as of z/OS V1R8 XL C/C++
    - IPA(LINK) defaults 65
    - IPA binary compatibility 66
- linkage editor control statements
  - pre-OS/390 modules
    - calls to COBOL routines 36
- linkage issues
  - as of V1R10 88
  - as of V1R9 with PTF UK31348 88
- listings
  - as of z/OS V1R6 C/C++
    - binding OS/390 modules 74
    - formats 74
    - binding OS/390 modules 103
    - format changes 25, 61, 87
    - formats 103
- Load Module Analyzer (LMA)
  - CICS processing
    - as of z/OS V1R9 136
- load modules
  - converting pre-OS/390 programs 38
- LOCALDEF utilities
  - as of z/OS V1R6 108
- LOCALE compiler option
  - and macro redefinitions
    - as of z/OS V1R9 XL C/C++ 94
- locale name
  - as of z/OS V1R9
    - \_\_LOCALE\_\_ macro 67
    - LOCALE compiler option 67
- localtime() 47
- long long data type
  - as of z/OS V1R7 XL C++
    - C99 standard macros 83
- long long macros
  - as of z/OS V1R7 XL C++
    - numeric conversion functions 83
- LP64 compiler option
  - as of z/OS V1R6 C/C++ 95
  - as of z/OS V1R8 XL C/C++
    - and GONUMBER compiler option 93
- LP64 environment restriction
  - as of z/OS V1R6 C/C++
    - with \_DEBUG\_FORMAT environment variable 69
- LSEARCH compiler option 28
  - as of z/OS V1R2 C/C++ 62

## M

- M compiler option
  - as of z/OS V1R11 67, 95
- macros
  - for LANGLVL(EXTENDED)
    - z/OS V1R7 XL C 27
- macro definition check
  - SQL coprocessor-based compilations
    - as of z/OS V1R10 XL C/C++ 140
- macro redefinitions
  - as of z/OS V1R7 XL C/C++
    - under LANGLVL(ANSI), LANGLVL(SAA), or LANGLVL(SAAL2) 93

- macro redefinitions (*continued*)
  - as of z/OS V1R7 XL C/C++ (*continued*)
    - under LANGLVL(EXTENDED) 94
- macro undefinition and redefinition
  - SQL coprocessor-based compilations
    - as of z/OS V1R10 XL C/C++ 140
- macros
  - \_OPEN\_SYS\_SOCKET\_IPV6
    - as of z/OS V1R7 XL C++ 82
  - as of z/OS V1R11
    - \_\_IBMCPP\_EXTENDED\_FRIEND 124
  - as of z/OS V1R6 XL C
    - \_LONG\_LONG 67
  - as of z/OS V1R9 XL C/C++
    - \_\_LOCALE\_\_ macro 94
  - for certain language levels
    - as of z/OS V1R7 XL C 25, 27, 62, 66, 90, 93
    - for LANGLVL(EXTENDED) V1R7 XL C 94
    - z/OS V1R7 XL C 66
    - for LANGLVL(EXTERNTEMPLATE) z/OS V1R11 125
- macros, standard
  - as of z/OS V1R7 XL C++
    - C99 support of 83
    - TARGET compiler option 83
- main programs, fetched
  - pre-OS/390 source code 21
- mainframe
  - education xvi
- maintenance level, determining 139
- mangled names
  - as of z/OS V1R3 C/C++ 34
- math functions
  - as of z/OS V1R9
    - IEEE754 109
- MEMLIMIT default value
  - as of z/OS V1R8 XL C/C++
    - 64-bit memory 99
    - 64-bit virtual memory 65
    - overriding 65, 99
    - setting 65, 99
- memory requirements
  - as of z/OS V1R8 XL C/C++ 29
  - as of z/OS V1R9 XL C/C++
    - IPA link step 70
    - IPA link step 29
    - as of z/OS V1R8 XL C/C++ 98
- message data sets
  - NATLANG runtime option 42, 61
- messages
  - CICS 134
  - CICS reason codes 134
  - contents 41
  - debug format
    - as of z/OS V1R6 C/C++ 61, 96
  - differences between C/370 and AD/Cycle C/370 V1R2 41
  - differences between C/370 and Language Environment 41
  - differences between pre-OS/390 and Language Environment runtime messages 41
  - macro redefinitions
    - as of z/OS V1R11 125

- messages (*continued*)
  - macro redefinitions (*continued*)
    - as of z/OS V1R7 XL C 25, 27, 62, 66, 90, 93, 94
  - MSGFILE runtime option 43
  - non-DLL compilations
    - as of z/OS V1R6 C/C++ 96
  - perror() 42
  - prefixes 41
  - specifying the national language
    - for 42, 61
  - strerror() 42
  - Unable to open DBRM file
    - as of z/OS V1R8 XL C 141
  - migration objectives and recommended approaches 117
  - mktime() 47
  - Model Tool support
    - as of OS/390 V2R10 C/C++ 70
  - MSGFILE runtime option
    - pre-OS/390 modules 43
  - multithreaded applications
    - binding OS/390 modules 71
  - MVS batch interface
    - as of z/OS V1R6 108
  - MVS/ESA V3
    - dumps 43

## N

- name lookups
  - as of z/OS V1R10 XL C++ 119
- name mangling
  - as of z/OS V1R3 C/C++ 34
  - as of z/OS V1R9 XL C/C++
    - and batch processing 98
- namespace pollution
  - as of z/OS V1R9 XL C++
    - IEEE 754 interface declarations 81
    - math.h 81
    - SQL coprocessor-based compilations
      - as of z/OS V1R10 139
  - as of z/OS V1R8 XL C/C++
    - handling 33, 101
- namespace pollution errors
  - SQL coprocessor-based compilations
    - handling, as of z/OS V1R10 139
- namespaces
  - as of z/OS V1R10
    - avoiding pollution of 139
  - as of z/OS V1R9 XL C++
    - <net/if.h> header file 100
    - avoiding pollution of 81
    - XPG 4.2 100
  - national language for runtime environment, specifying 42
  - NATLANG runtime option 42
    - C/370 equivalent 43
    - message data sets 61
  - navigation
    - keyboard 145
- new
  - pre-OS/390 source code
    - array format 20

- new, array version
  - as of z/OS V1R2 C/C++
    - avoiding syntax errors 129
    - pre-OS/390 source code 20
- non-DLL compilations
  - as of z/OS V1R6 C/C++ 96
- NONIPSTACK runtime option
  - Language Environment
    - equivalent 43
- NORENT compiler option 71
  - as of OS/390 V2R9
    - variables 68
- NOSPIE runtime option
  - running pre-OS/390 programs 44
- NOSTAE runtime option
  - running pre-OS/390 programs 44
- Notices 149
- NULL assignments
  - pre-OS/390 source code 21
- numeric conversion functions
  - as of z/OS V1R7 XL C++
    - long long macros 83
    - C99 support 111

**O**

- object models, supported
  - as of z/OS V1R6 C/C++ 102
- OE compiler option
  - as of z/OS V1R2 C/C++ 62
- OMVS compiler option
  - alternative as of z/OS V1R2
    - C/C++ 26
  - as of z/OS V1R2 C/C++
    - alternative 62
- optimization
  - as of OS/390 V2R6 C/C++ 62
- OPTIMIZE compiler option 28
  - as of z/OS V1R5 C/C++
    - OPT(3) 68
- OS/390 behavior
  - retaining 73
- OS/390 migration issues
  - Language Environment
    - customization 74
- OS/390 migrations
  - JCL procedures 70
- OS/390 modules
  - as of z/OS V1R7 XL C/C++
    - bind step 71
- OS/390 programs
  - improving performance 69
- OS/390 V1R4
  - Database Access Class Library utility
    - removal of support 75
- OS/390 V2R10
  - removal of Model Tool support 70
  - ROSTRING compiler option 68
  - System Object Model (SOM)
    - removal of support 75
- OS/390 V2R6
  - optimization level mapping and
    - listing content 62
- OS/390 V2R9
  - #pragma leaves 69
  - #pragma reachable 69
  - #pragma variable 68

- OS/390 V2R9 (*continued*)
  - enclaves
    - abnormal terminations 44
    - NORENT compiler option 68
    - variables 68
  - overflow processing
    - and ARCH option 59
    - as of z/OS V1R6 C/C++
      - and ARCHITECTURE level 63
      - and data conversions 63
    - OS/390 source code
      - examples 59
  - overload ambiguities
    - as of z/OS V1R2 C++
      - avoiding 127
  - overloads of standard math functions
    - as of z/OS V1R2 C++
      - avoiding exceptions 127

**P**

- packed structures and unions
  - assignment restrictions
    - migration from pre-OS/390 21
  - DSECT header files
    - migration from pre-OS/390 21
- PDF documents xv
- PDS 51
- PDSE 51
- performance improvements
  - as of z/OS V1R9
    - IEEE754 math functions 109
- performance, improving
  - as of z/OS V1R9 XL C/C++
    - very large applications 70
  - very large applications
    - as of z/OS V1R8 XL C 29, 98
  - when recompiling OS/390
    - programs 69
- perror() 42
- PL/1 interlanguage calls
  - pre-OS/390 modules 35
- PL/I interlanguage calls 35
- pointer casting
  - as of z/OS V1R2 C/C++
    - anti-aliasing rule 64
- pointer incompatibilities
  - pre-OS/390 source code 21
- portability
  - to or from AIX
    - as of z/OS V1R6 C/C++ 95
- POSIX compliance
  - as of z/OS V1R5
    - changes to putenv() 107
  - as of z/OS V1R5 C/C++
    - putenv() function 82
  - POSIX compliance 73
  - retaining OS/390 behavior 73
- potential linkage issues
  - as of V1R10 88
  - as of V1R9 with PTF UK31348 88
- pragma
  - enum
    - as of z/OS V1R2 C/C++ 64
  - pack
    - DSECT header files 21

- pragma (*continued*)
  - runopts
    - pre-OS/390 source code 19
    - variable
      - as of OS/390 V2R10 C/C++ 68
- pragmas
  - as of z/OS V1R2 XL C++
    - pack(2) 84
  - as of z/OS V1R7 XL C/C++
    - variable 71
  - binding OS/390 modules 71
  - changes in behavior of variables 71
  - leaves
    - as of OS/390 V2R9 69
  - reachable
    - as of OS/390 V2R9 69
  - runopts 43
- pre-OS/390 applications
  - runtime
    - compatibility issues 45
- pre-OS/390 source code
  - NULL assignments 21
  - pointer incompatibilities 21
- program masks
  - CICS applications
    - pre-OS/390 source code 22
  - pre-OS/390 source code 22
  - System Programming C
    - pre-OS/390 source code 22
- pselect() interface
  - as of z/OS V1R11 XL C++ 100
- PSW mask 22
- putenv()
  - as of z/OS V1R5
    - and POSIX compliance 107
- putenv() function
  - as of z/OS V1R5 C/C++ 82

**R**

- realloc() function
  - migration from pre-OS/390 49
  - pre-OS/390 source code
    - modification 49
- recommended approaches for migration
  - objectives 117
- reentrancy
  - as of OS/390 V2R10 C/C++
    - #pragma variable 68
  - as of OS/390 V2R9
    - #pragma variable 68
  - as of z/OS V1R7 XL C/C++
    - binding OS/390 modules 71
- region size
  - as of z/OS V1R9 XL C/C++
    - default 70
- release changes and migration issues 3, 5
- relink requirements
  - ctest() 34
- REPORT runtime option
  - Language Environment
    - equivalent 43
- REPORT with #pragma runopts
  - pre-OS/390 source code 19

- resolution of conflicts between options
    - and pragmas
    - as of z/OS V1R7 XL C/C++ 91
  - resource allocation
    - and memory management
    - pre-OS/390 source code 49
  - return codes
    - control of processing
    - as of z/OS V1R10 97
    - specifying maximum acceptable
    - as of z/OS V1R10 97
  - return codes differences
    - between C/370 and Language Environment 41
  - ROCONST compiler option
    - default as of z/OS V1R2 C/C++ 68
  - ROSTRING compiler option
    - as of z/OS V1R2 C/C++ 68
  - RPTSTG runtime option
    - C/370 equivalent 43
  - rules of precedence
    - user exits 33
  - runtime behavior, OS/390
    - retaining for the greatest number of items 73
  - runtime behavior, pre-OS/390
    - retaining for the greatest number of items 41
  - runtime behavior, previous
    - daylight saving time 107
    - internal timing algorithm 107
    - retaining earlier IEEE754 math functions 107
    - retaining for the greatest number of items 106
  - runtime compatibility issues
    - pre-OS/390 applications 45
  - runtime libraries
    - C/370, under CICS 133
    - C99 standard
      - floating-point notation 110
      - floating-point special values 111
  - Runtime Library Extensions
    - earlier z/OS C/C++ source code 79
    - OS/390 source code 59
    - pre-OS/390 source code 19
  - runtime options
    - ABTERMENC 44
      - abnormal terminations of enclaves 44
    - C/370 V2 compiler to z/OS V1R9 C compiler 41
    - ending options list 43
    - HEAP 44
      - C/370 V2 compiler to z/OS V1R9 C compiler 44
    - ISAINC
      - Language Environment equivalent 43
    - ISASIZE
      - Language Environment equivalent 43
    - LANGUAGE
      - Language Environment equivalent 43
    - MSGFILE 43
      - passing to program 43
  - runtime options (*continued*)
    - pre-OS/390 41
    - REPORT
      - Language Environment equivalent 43
    - slash (/) 43
    - SPIE
      - Language Environment equivalent 43
    - SPIE|NOSPIE 44
    - STAE
      - Language Environment equivalent 43
    - STAE|NOSTAE 44
    - TRAP 44
  - runtime options, specifying in JCL 29
- S**
- scanf()
    - as of z/OS V1R9 XL C++
      - impact of DFP size modifiers, source code modifications 82
  - SCEERUN library module 46
    - environment initialization 47
  - SCLBH data sets 30
  - scope information
    - handling
      - as of z/OS V1R9 XL C/C++ 81
  - SEARCH compiler option 28
    - as of z/OS V1R2 C/C++ 62
  - sending comments to IBM xvii
  - setlocale() function
    - as of z/OS V1R6 108
  - shortcut keys 145
  - SIBMLINK library module 46
    - environment initialization 47
  - SIGFPE exceptions 48
    - CICS applications
      - pre-OS/390 source code 22
    - pre-OS/390 binder error 48
    - pre-OS/390 source code 22
    - System Programming C
      - pre-OS/390 source code 22
  - SIGINT exception
    - changes from C/370 V2 48
  - SIGTERM exception
    - changes from C/370 V2 48
  - SIGUSR1 exception
    - changes from C/370 V2 48
  - SIGUSR2 exception
    - changes from C/370 V2 48
  - sizeof operator
    - pre-OS/390 source code 20
  - SOM compiler option
    - as of OS/390 V2R10
      - removal of SOM support 75
  - source code
    - pre-OS/390 compiler to z/OS V1R9 XL C/C++ 19
    - source code incompatibilities
      - with earlier releases of the z/OS C/C++ compiler 79
      - with OS/390 programs 59
    - source code modifications
      - as of z/OS V1R9 XL C++
        - impact of DFP size modifiers 82
  - SPIE runtime option
    - Language Environment
      - equivalent 43
      - running pre-OS/390 programs 44
  - SPIE with #pragma runopts
    - pre-OS/390 source code 19
  - SQL compiler option
    - as of z/OS V1R10 XL C 139
    - as of z/OS V1R9 XL C 139
  - SQL coprocessor-based compilations
    - as of z/OS V1R10
      - namespace pollution 139
  - SQL Universal Database
    - requesting DB2 services 139
    - z/OS V1R5 XL C — z/OS V1R8 XL C 139
  - SRCMSG compiler option
    - as of z/OS V1R2 C/C++ 26, 62
  - STACK runtime option
    - as of z/OS V1R2 C/C++ 45
    - C/370 equivalent 43
    - parameters 45
    - STACK defaults 45
  - STAE runtime option
    - Language Environment
      - equivalent 43
      - running pre-OS/390 programs 44
  - STAE/SPIE with #pragma runopts
    - pre-OS/390 source code 19
  - Standard C++ compliance 99
    - array new with user-defined global new operator
      - pre-OS/390 20
    - as of z/OS V1R2
      - access checking 126
      - access checking (C++ only) 126
      - CCN5193 exception 127
      - class type definitions 126
      - exception handling 126
      - exceptions 126
      - type definitions 126, 127
    - as of z/OS V1R2 C/C++
      - syntax error with array new 129
    - as of z/OS V1R2 C++
      - ambiguous overloads 127
      - effect on friend declarations 121
    - as of z/OS V1R7 XL C++ 80
    - as of z/OS V1R9
      - CCN5413 exception 126
      - class access checking 126
    - as of z/OS V1R9 XL C++ 80
    - effect on exception handling 122
    - implicit integer types
      - as of z/OS V1R2 C++ 120
    - scoping for-loop initializer declarations
      - as of z/OS V1R2 C++ 120
    - statically initialized objects, destruction of 120
    - user-defined conversions 128
  - Standard C++ compliance and friend declarations in class member lists
    - as of z/OS V1R2 C++ 121
  - Standard C++ I/O Stream Library and UNIX System Laboratories Complex Mathematics Library 80

- standard math functions
  - as of z/OS V1R2 C++
  - ambiguous overloads 127
- standard stream support
  - under CICS 134
- static code 46
- statically initialized objects, destruction
  - of 120
- STATICINLINE compiler option
  - default as of z/OS V1R2 C/C++ 68
- stderr 43
  - output handling under CICS 135
- strerror() 42
- symbolic names
  - resolution as of V1R9 89
- SYSERR ddname
  - pre-OS/390 modules 43
- SYSLIB compiler option
  - alternative as of z/OS V1R2 C/C++ 26
  - as of z/OS V1R2 C/C++
  - alternative 62
- SYSMSGSGS ddname 42
- SYSPTH compiler option
  - alternative as of z/OS V1R2 C/C++ 26
  - as of z/OS V1R2 C/C++
  - alternative 62
- SYSPRINT ddname
  - pre-OS/390 modules 43
- system header files
  - type declarations
    - as of z/OS V1R7 XL C++ 92
- System Object Model
  - as of OS/390 V2R10 C/C++ 62
  - no longer supported 62
- System Object Model (SOM)
  - as of OS/390 V2R10
  - removal of SOM support 75
- System Programming C (SPC) facility
  - applications built with
    - EDCXSTRX 22
    - CEEEV003 22
    - EDCXV 22
    - source changes 22
- SYSTEM ddname
  - pre-OS/390 modules 43

## T

- TARGET compiler option
  - and binder features 102
  - as of z/OS V1R6 C/C++ 96
  - as of z/OS V1R7 XL C++
    - C99 standard macros 83
  - as of z/OS V1R8 XL C/C++ 102
  - as of z/OS V1R9 XL C/C++ 69
  - earliest release that can be targeted
    - as of z/OS V1R12 XL C/C++ 95
  - targeting an earlier release
    - as of z/OS V1R12 XL C/C++ 95
    - as of z/OS V1R8 XL C/C++ 102
  - technical support xvi
- TEMPLATEDEPTH compiler option
  - as of z/OS V1R13 XL C/C++ 95

- templates
  - as of z/OS V1R9 XL C++
  - name lookup exceptions 90
- terminate\_\_3stdFv binder error
  - message 101
- TEST compiler option
  - as of z/OS V1R6 C/C++ 29, 69
  - PATH suboption
    - as of z/OS V1R6 C/C++ 29
- thread processing
  - as of z/OS V1R8 XL C/C++
  - processor release 80
  - processor release
    - as of z/OS V1R8 107
- time zone issues 47
- time.h header file
  - as of z/OS V1R9
  - localtime() function 107
- TRAP runtime option
  - C/370 equivalent 43
  - running pre-OS/390 programs 44
- TSO localedef utility interface
  - as of z/OS V1R6 108
- TUNE compiler option
  - as of z/OS V2R1 C/C++
  - default 69
- twobyte packed data alignment
  - as of z/OS V1R2 XL C++
  - unexpected C++ output 84
- type definitions
  - as of z/OS V1R2
  - avoiding errors 126
- typographical conventions x

## U

- ulimit command
  - as of z/OS V1R8 XL C/C++
  - MEMLIMIT system parameter 65, 99
- unexpected results
  - as of z/OS V1R9 XL C++
  - impact of DFP size modifiers on
    - fprintf/fscanf results 106
- unset()
  - effect upon behavior of fflush() 53
  - effect upon behavior of fgetpos() 53
  - effect upon behavior of fseek() 53
- unhandled conditions
  - changes from C/370 V2 48
- Unicode character translation
  - and #pragma comment strings
    - as of z/OS V1R10 XL C/C++ 63
- UNIX System Laboratories
  - and Standard C++ I/O Stream
    - libraries 75
- UNIX System Laboratories Complex
  - Mathematics Library
    - and Standard C++ I/O Stream
      - Library 80
      - earlier z/OS C/C++ source code 79
      - OS/390 source code 59, 75
      - pre-OS/390 source code 19
  - UNIX System Laboratories I/O Stream
    - Library
      - earlier z/OS C/C++ source code 79
      - OS/390 source code 59, 75

- UNIX System Laboratories I/O Stream
  - Library (*continued*)
    - pre-OS/390 source code 19
- UNIX System Services files, support
  - of 33
- unrolling loops
  - as of z/OS V1R7 XL C/C++ 83
- USEPCH compiler option
  - as of z/OS V1R2 C/C++ 62
- user exits
  - as of z/OS V1R5
    - CEEBDATX 42
    - CEEBDATX 33
    - CEEBXITA library module 33
    - CEECDATX 133
    - IBMBXITA library module 33
- user interface
  - ISPF 145
  - TSO/E 145
- user name spaces
  - pre-OS/390 modules 36
- user-defined conversions
  - avoiding exceptions 128
- USERLIB compiler option
  - alternative as of z/OS V1R2 C/C++ 26
  - as of z/OS V1R2 C/C++
  - alternative 62
- USERPATH compiler option
  - alternative as of z/OS V1R2 C/C++ 26
  - as of z/OS V1R2 C/C++
  - alternative 62
- using directive
  - as of z/OS V1R10 XL C++ 119

## V

- variable mode
  - as of z/OS V1R9
  - C99 compliance 109
- variables
  - as of z/OS V1R7 XL C/C++
  - binding OS/390 modules 71
  - reentrant 71
- very large applications
  - as of z/OS V1R9 XL C/C++
  - IPA link step 70
  - macro redefinition 70
  - IPA Link step
    - as of z/OS V1R8 XL C 29, 98
- virtual functions
  - declaring and calling
    - as of z/OS V1R6 C/C++ 84

## W

- WSIZEOF compiler option
  - pre-OS/390 source code 20

## X

- XL C DB2 coprocessor 139
- XL C/C++ compiler invocations
  - as of z/OS V1R6 C/C++ 95

- xlC configuration file
  - as of z/OS V1R7 XL C/C++
    - customizing 91
- xlC invocation
  - as of z/OS V1R7 XL C/C++
    - resolution of conflicts between options and pragmas 91
- xlC utility
  - and TEMPINC 95
  - as of z/OS V1R10
    - return-code processing 97
  - as of z/OS V1R7 XL C/C++ 97
  - source code changes 97
  - xlC command 95
  - xlC command 95
  - xlC++ command 95
- XPLINK compiler option
  - as of z/OS V1R6 C/C++ 95
- XPLINK runtime option
  - C/370 equivalent 43

## Z

- z/OS Basic Skills information center xvi
- z/OS UNIX System Services
  - as of z/OS V1R8 XL C/C++
    - ulimit command 65, 99
- z/OS V1R10
  - AMODE 64 applications 105
  - diagnostic changes
    - potential linkage issues 88
  - HEAPOOLS runtime option 105
  - listings show compiler substitution variables 88
  - namespace pollution errors 139
  - PTF UK31348 88
  - requesting DB2 services 139
  - return-code processing
    - options 97
  - SQL coprocessor-based
    - compilations 139
    - macro definition check, performing 140
    - macro undefinition and redefinition 140
  - xlC utility
    - return-code processing 97
- z/OS V1R10 XL C/C++
  - #pragma comment and ASCII 63
  - ASCII users 63
- z/OS V1R10 XL C++
  - name lookups 119
  - using directive 119
- z/OS V1R11
  - \_POSIX\_C\_SOURCE macro 100
  - C++11 126
  - corrections in escape sequence
    - encoding 88
  - extendedfriend 124
  - feature testing 100
  - friend declaration 124
  - header files 100
  - LANGLVL(EXTENDED0X) compiler option 94
  - LANGLVL(EXTERNTEMPLATE) compiler option 125
  - M compiler option 67, 95
  - z/OS V1R11 (*continued*)
    - macro redefinitions 125
    - WARN0X compiler option 126
- z/OS V1R12
  - earliest release that can be targeted 95
  - LANGLVL(AUTOTYPEDEDUCTION) compiler option 123
  - LANGLVL(C99LONGLONG) compiler option 123
  - LANGLVL(C99PREPROCESSOR) compiler option 123
  - LANGLVL(DECLTYPE) compiler option 124
  - LANGLVL(DELEGATINGCTORS) compiler option 124
  - LANGLVL(EXTENDEDINTEGERSAFE) compiler option 125
  - LANGLVL(INLINENAMESPACE) compiler option 125
  - LANGLVL(STATIC\_ASSERT) compiler option 126
  - LANGLVL(VARIADICTEMPLATES) compiler option 126
  - RESTRICT 95
  - SEVERITY 95
  - TARGET compiler option 95
- z/OS V1R13
  - CHECKOUT compiler option 91
  - FLAG compiler option 93
  - GENASM compiler option 93
- z/OS V1R13 XL C/C++
  - ARGPARSE compiler option 63
  - DSAUSER compiler option 92
  - TEMPLATEDEPTH compiler option 95
- z/OS V1R2
  - #pragma enum 64
  - #pragma variable 68
  - ambiguous overloads 127
  - ANSI-aliasing rule 64
  - as of z/OS V1R2 C/C++
    - HALTONMSG compiler option 64
  - batch processing
    - alternative 30
    - SYSLIB concatenation 30
  - C support 27
  - C++ exception handling 122
  - CC EXEC invocation changes 33
  - CHECKOUT(CAST) compiler option 64
  - compiler options, no longer supported 62
  - cv-qualification 122
  - DECK compiler option 25
  - destruction of statically initialized objects before and after ISO/IEC 14882:2003(E) compliance 120
  - DIGRAPH compiler option
    - default 64
  - enumeration types
    - controlling size of 26
    - enumeration types, controlling size of 64
  - ENUMSIZE() compiler option 64
- z/OS V1R2 (*continued*)
  - friend declarations in class member lists 121
  - friend declarations, visibility of 121
  - HWOPTS compiler option 26
  - implicit integer types and Standard C++ compliance 120
  - include files, finding 26
  - INFO compiler option 27
  - INLINE compiler option 27
    - defaults 64
  - ISO standard C++ compliance 115
  - LANGLVL(COMPAT) compiler option 26
  - LANGLVL(OLDMATH) compiler option 127
  - library file searches 30
  - OMVS compiler option 26
  - pack(2) 84
  - pointer casting 64
  - ROSTRING compiler option 68
  - scoping for loops 120
  - SRCMSG compiler option 26
  - STACK runtime option 45
  - Standard C++ compliance 122
    - C++ class access errors 126
  - STATICINLINE compiler option 68
  - syntax error with array new 129
  - SYSLIB compiler option 26
  - SYSLIB DD cards to remove 30
  - twobyte packed data alignment 84
  - unexpected C++ output 84
  - USERLIB compiler option 26
- z/OS V1R3
  - #pragma map 34
  - C++ class names 34
  - external variable names 34
  - name mangling 34
- z/OS V1R5
  - \_EDC\_PUTENV\_COPY environment variable 82
  - abnormal termination exit routine 42
  - batch processing 42
  - CEEBDATX 42
  - changes to putenv() 107
  - compiling OS/390 applications 70
  - destruction of statically initialized objects before and after ISO/IEC 14882:2003(E) compliance 120
  - JCL procedures 70
    - Language Environment customization 99
  - locale name 68
  - OPTIMIZE compiler option 68
  - POSIX compliance 82, 107
  - putenv() function 82
  - requesting DB2 services 139
- z/OS V1R5 C/C++, earlier than
  - JCL procedures
    - Language Environment customization 99
- z/OS V1R6
  - \_DEBUG\_FORMAT environment variable 74, 96, 103
  - @euro locale 108
  - @preeuro locale 108



- z/OS V1R6 (*continued*)
  - alignment incompatibilities
    - between object models 102
  - ARCHITECTURE level and overflow processing 63
  - batch processing 108
  - binding OS/390 modules 74
  - C support 27, 64
  - c89 utility 74, 96, 103
  - c89 utility and `_DEBUG_FORMAT` environment variable 61
  - CHECKOUT compiler option 64
  - COMPAT compiler option 96
  - data types 67
  - declaring and calling virtual functions 84
  - dynamic binding 84
  - EEC default currency 108
  - INFO compiler option 27, 64
  - interlanguage calls (ILC)
    - with `#pragma pack(2)` 102
  - ISO standard C++ compliance
    - determining level supported by compiler 115
  - Language Environment
    - customization 108
  - LC\_MONETARY information 108
  - listings 74
  - LOCALDEF utilities 108
  - long long 67
  - LP64 compiler option 95
  - MVS batch interface 108
  - object module incompatibilities
    - with `#pragma pack(2)` 102
  - pre-OS/390 modules and language libraries 36
  - pre-OS/390 modules and user name spaces 36
  - requesting DB2 services 139
  - setlocale() function 108
  - TARGET compiler option 96
  - TEST compiler option 29, 69
  - TSO localedef utility interface 108
  - xlC command 95
  - xlC command 95
  - xlC++ command 95
  - XPLINK compiler option 95
- z/OS V1R7 64
  - `_OPEN_SYS_SOCKET_IPV6` macro 100
  - `_OPEN_SYS_SOCKET_IPV6` macro and `netinet/in.h`
    - new definitions exposed 82
  - `-qpluscmt` command option
    - when to override 97
  - `#pragma unroll()` 83
  - C99 support 83
  - CICS statement translation
    - options 133
  - CMDOPTS compiler option 91
  - comments, using 97
  - enumeration types
    - controlling size of 26
  - `ENUMSIZE(SMALL)` 92
  - feature testing 100
  - for loops 83
  - header files 100
  - LANGLVL compiler option 90
- z/OS V1R7 (*continued*)
  - and macro redefinitions 93, 94
  - LANGLVL(ANSI) compiler option 66
  - LANGLVL(EXTENDED) compiler option 66
  - LANGLVL(LONGLONG) compiler option 83
  - LANGLVL(SAA) compiler option 66
  - LANGLVL(SAA2) compiler option 66
  - Language Environment services 100
  - macro redefinition 90
  - macro redefinitions 66
    - LANGLVL compiler option 93, 94
  - numeric conversion functions 83
  - protected enumeration types in system header files 92
  - reentrant variables with NORENT
    - binding OS/390 modules 71
    - JCL procedures 71
  - requesting DB2 services 139
  - resolution of conflicts between options and pragmas 91
  - Standard C++ compliance 80
  - TARGET compiler option 83
  - under CICS 134
  - unrolling loops 83
  - xlC configuration file 91
- z/OS V1R8
  - `_PVERSION` environment variable 102
  - 64-bit processing 93
  - 64-bit virtual memory 65
  - binder errors
    - namespace pollution 33, 101
  - c89 utility
    - binder, invoking 71
  - c89 utility and COMPAT binder option 102
  - errors binding earlier z/OS C/C++ programs
    - namespace pollution 101
  - errors binding pre-OS/390 programs
    - namespace pollution 33
  - GONUMBER compiler option 93
  - internal timing algorithm 107
  - IPA compiler option 98
  - IPA link step 98
  - IPA(LINK)
    - 64-bit memory 99
    - MEMLIMIT default value 99
  - IPA(LINK) compiler option 65
    - link step defaults 65
  - JCL procedures 65
  - library functions 80
  - memory requirements 98
  - performance, improving
    - very large applications 98
  - processor release 80
  - requesting DB2 services 139
  - setting MEMLIMIT value 65
  - targeting an earlier release 102
  - thread processing 80
- z/OS V1R9
  - `_LOCALE_` macro 67
  - `_ICONV_MODE` environment variable
    - user-defined conversion tables 98
- z/OS V1R9 (*continued*)
  - `_XOPEN_SOURCE_EXTENDED` macro 100
  - `<net/if.h>` header file 100
  - array index definitions 89
  - as of z/OS V1R9 XL C/C++
    - default region size 97
  - batch processing and name mangling
    - ILP32 compiler option 98
  - C99 support 109
  - CICS processing
    - binary converter tables 136
    - HFS 136
    - iconv() changes and
      - CEECCSD.COPY and CEECCSDX.COPY files 136
    - Load Module Analyzer (LMA) 136
    - Unicode converters 136
    - using AFP registers 136
  - Communications Server
    - information 81
  - default daylight saving time 107, 108
  - DFP
    - size modifiers 82, 106
  - diagnostic changes
    - potential linkage issues 88
  - error messages
    - name lookup exceptions 90
  - feature test macros and system header files 81
  - feature testing 100
  - FLOAT(IEEE) compiler option 109
  - getnameinfo() function 81
  - IEEE 754 interface declarations 81
  - IEEE754 math functions 109
  - ILP32 compiler option
    - batch processing and name mangling 98
  - initialization incompatibility with C/370 modules 46
  - IPA compiler option 70
  - ISO standard C++ compliance 115
  - JCL procedures
    - assembly listings 90
    - user-defined conversion tables 98
  - Language Environment services 100
  - library functions 80
  - LOCALE compiler option 67
    - and macro redefinitions 94
  - locale name 67
  - macro redefinitions
    - LOCALE compiler option 94
  - PTF UK31348 88
  - pthread\_yield() function 80
  - region size, default 70
  - requesting DB2 services 139
  - scope information 81
  - Standard C++ compliance 80
  - symbolic names 89
  - TARGET compiler option 69
  - templates 90
  - variable mode 109
- z/OS V2R1
  - ARCHITECTURE default 63
  - ARCHITECTURE level and SYSSTATE
  - ARCHLVL statement 63

z/OS V2R1 (*continued*)  
C++11 compiler option 122  
IPA compiler option 93  
LANGLVL(C1XNORETURN) compiler  
option 123  
LANGLVL(CONSTEXPR) compiler  
option 124  
LANGLVL(DEFAULTANDDELETE)  
compiler option 124  
LANGLVL(EXPLICITCONVERSIONOPERATORS)  
compiler option 124  
LANGLVL(EXTC1X) compiler  
option 93  
LANGLVL(REFERENCECOLLAPSING)  
compiler option 125  
LANGLVL(RIGHTANGLEBRACKET)  
compiler option 125  
LANGLVL(RVALUEREFERENCES)  
compiler option 126  
LANGLVL(SCOPEENUM) compiler  
option 126  
TUNE default 69  
zFS files, support of 70





Product Number: 5650-ZOS

Printed in USA

GC14-7306-01

