z/OS

C/C++
Compiler and Run-Time Migration Guide

GC09-4913-00

z/OS

C/C++
Compiler and Run-Time Migration Guide

GC09-4913-00

Note!
Before using this information and the product it supports, be sure to read the information in “Notices” on page 111.

Second Edition (October 2001)

This edition applies to Version 1 Release 2 Modification 0 of z/OS C/C++ (5694-A01) and to all subsequent releases and modifications until otherwise indicated in new editions. This edition replaces SC09-4763-00. Make sure that you use the correct edition for the level of the program listed above. Also, ensure that you apply all necessary PTFs for the program.

Order publications through your IBM® representative or the IBM branch office serving your location. Publications are not stocked at the address below. You can also browse the books on the World Wide Web by clicking on ″The Library″ link on the z/OS home page. The web address for this page is
`http://www.ibm.com/servers/eserver/zseries/zos/bkserv`

IBM welcomes your comments. You can send your comments by mail to the following address:

IBM Canada Ltd. Laboratory
Information Development
B3/KB7/8200/MKM
8200 Warden Avenue
Markham, Ontario Canada
L67 1C7

If you send comments, include the title and order number of this book, and the page number or topic related to your comment. When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Part 1. Introduction

This part provides answers to some common migration questions.

Note that throughout this book, the short form of a product's version and release (VxRx) is used. For example, this book refers to *OS/390 Version 2 Release 4 C/C++* as *OS/390 V2R4 C/C++*. In addition, assume that the modification level of any referenced product is 0 (zero) unless specifically indicated. For example, this book refers to *AD/Cycle C/370 Version 1 Release 1 Modification 1* as *AD/Cycle C/370 V1R1M1*.

Chapter 1. Locating your Migration Path

This book discusses the implications of migrating applications from each of the compilers and libraries listed in Table 2 on page 4 to the z/OS V1R2 C/C++ product. To find the section of the book that applies to your migration, see “How This Book Is Organized”.

Use this book to help determine what must be done to continue to use existing source code, object code, and load modules, and to be aware of differences in behavior between products that may affect your migration. In most situations, existing well-written applications can continue to work without modification.

This book does not discuss all of the enhancements that have been made to the z/OS V1R2 C/C++ compiler and z/OS Language Environment V1R2. This book does not show how to change an existing C program so that it can use C++. For a list of books that provide information about the z/OS V1R2 C/C++ compiler and its class libraries, debugger, and utilities, refer to “Bibliography” on page 115. For a description of some of the differences between C and C++, see *C/C++ Language Reference*.

In this book, references to the products listed in the first column of Table 1 also apply to the products in the second column.

Table 1. Product References

References To These Products	Also Apply To These Products
LE/370 R3	MVS/ESA™ SP™ V5R1 OpenEdition®, AD/Cycle® C/370™ Language Support Feature
Language Environment® R4	C/C++ Language Feature of MVS/ESA SP V5R2M0
Language Environment R5	C/C++ Language Feature of MVS/ESA SP V5R2M2
C/MVS™ V3R2 compiler	C component of the C/C++ for MVS/ESA V3R2 compiler
C++/MVS V3R2 compiler	C++ component of the C/C++ for MVS/ESA V3R2 compiler
OS/390 V1R1	IBM C/C++ for MVS™ V3R2 compiler and Language Environment R5
OS/390 V2R10	IBM z/OS C/C++ V1R1 compiler

How This Book Is Organized

- Part 1 contains some general answers to common migration questions.
- Part 2 describes the considerations for migrating from one of the following:
  - The IBM C/370 V1 or V2R1 compiler and the IBM C/370 V1 or V2 library
  - The IBM SAA® AD/Cycle C/370 V1R2 compiler and the IBM C/370 V2R2 library
- Part 3 describes the considerations for migrating from one of the following compilers, and any release of z/OS Language Environment:
  - The AD/Cycle C/370 compilers
  - The MVS C/C++ V3 compilers
  - The OS/390 V1R1 C/C++ compiler
- Part 4 describes the considerations for migrating from one of the following:
  - OS/390 V1R2 C/C++

  - OS/390 V1R3 C/C++
  - OS/390 V2R4 C/C++
  - OS/390 V2R5 C/C++
  - OS/390 V2R6 C/C++
  - OS/390 V2R7 C/C++
  - OS/390 V2R8 C/C++
  - OS/390 V2R9 C/C++
  - OS/390 V2R10 C/C++
  - z/OS V1R1 C/C++
- Part 5 describes migration issues that may arise when using the z/OS V1R2 C/C++ compiler, which is fully compliant with the ISO C++ 1998 Standard.

A History of Compilers and Libraries

Table 2 lists the versions of the C and C++ compilers and run-time libraries in the order in which they were first released. Use this table to help determine which changes described in this book apply to your migration.

Table 2. A History of Compilers and Libraries

Short Name	Product Number	GA Date	Description	Service Status
C/370 V1R1	5688-040 5688-039	1988 1988	C/370 V1R1 Compiler C/370 V1R1 Library	Service discontinued Service discontinued
C/370 V1R2	5688-040 5688-039	1989 1989	C/370 V1R2 Compiler C/370 V1R2 Library	Service discontinued Service discontinued
C/370 V2R1	5688-187 5688-188	1991 1991	C/370 V2R1 Compiler C/370 V2R1 Library	
AD V1R1	5688-216	1991	AD/Cycle C/370 V1R1 Compiler, follow-on to C/370 V2R1 Compiler.	Service discontinued
LE V1R1	5688-198	1991	LE/370 V1R1 Library, first release of Language Environment/370; follow-on to C/370 V2R1 Library.	Service discontinued
LE V1R2	5688-198	1992	LE/370 V1R2 Library	Service discontinued
AD V1R2	5688-216	1994	AD/Cycle C/370 V1R2 Compiler: • Runs on either LE V1R3 or C/370 V2R2 • Generates code for either LE V1R3 or C/370 V2R2	
LE V1R3	5688-198	1994	LE/370 V1R3 Library, also shipped as part of MVS/ESA SP 5.1 OpenEdition AD/Cycle C/370 Language Support Feature.	Service discontinued
C/370 V2R2	5688-188	1994	C/370 V2R2 Library. Follow-on to the C/370 V2R1 Library, intended to help customers migrate to LE/370.	

Table 2. A History of Compilers and Libraries (continued)

Short Name	Product Number	GA Date	Description	Service Status
C/C++MVS V3R1	5655-121	1995	C/C++ for MVS/ESA V3R1 Compilers, follow-on to AD V1R2 Compiler. First release of C++ on MVS.	Service discontinued
LE V1R4	5688-198	1995	LE V1R4 Library for MVS & VM, also shipped as the MVS/ESA SP 5.2.0 C/C++ Language Support Feature.	Service discontinued
C/C++/ MVS V3R2	5655-121	1995	C/C++ for MVS/ESA V3R2 Compilers	
LE V1R5	5688-198	1995	LE V1R5 Library for MVS & VM, also shipped as part of MVS/ESA SP 5.2.2 C/C++ Language Support Feature.	
OS/390 R1	5645-001	March 1996	OS/390 R1 includes the C/C++ for MVS/ESA V3R2 Compilers and the OS/390 V1R1 Language Environment.	end of service Jan 31, 2001
OS/390 R2	5645-001	Sept 1996	OS/390 R2 C/C++ is the follow-on to OS/390 R1 C/C++, and includes new optimization options to improve the execution-time performance of C code. OS/390 V1R2 Language Environment comes with OS/390 V1R2.	end of service Jan 31, 2001
OS/390 R3	5645-001	March 1997	OS/390 R3 C/C++ is the follow-on to OS/390 R2 C/C++, and includes new optimization options to improve the execution-time performance of C++ code. OS/390 V1R3 Language Environment comes with OS/390 V1R3.	end of service Mar 31, 2001
OS/390 V2R4	5647-A01	Sept 1997	OS/390 V2R4 C/C++ is the follow-on to OS/390 R3 C/C++, and includes performance improvements for DLLs, conversion of character string literals, and support for the Program Management Binder. OS/390 V2R4 Language Environment comes with OS/390 V2R4.	end of service Mar 31, 2001
OS/390 V2R5	5647-A01	March 1998	OS/390 V2R5 C/C++ is functionally equivalent to OS/390 V2R4 C/C++.	end of service Mar 31, 2001
OS/390 V2R6	5647-A01	Sept 1998	OS/390 V2R6 C/C++ is the follow-on to OS/390 V2R4 C/C++. It includes support for the IEEE binary floating-point and the long long data types, improvements to the handling and format of packed decimal numbers in C++, and the `TARGET(OSV1R2)` suboption. OS/390 V2R6 Language Environment comes with OS/390 V2R6.	
OS/390 V2R7	5647-A01	March 1999	The compiler is functionally equivalent to the OS/390 V2R6 C/C++ compiler. OS/390 V2R7 Language Environment comes with OS/390 V2R7.	

Table 2. A History of Compilers and Libraries (continued)

Short Name	Product Number	GA Date	Description	Service Status
OS/390 V2R8	5647-A01	Sept 1999	The compiler is functionally equivalent to the OS/390 V2R6 C/C++ compiler. OS/390 V2R8 Language Environment comes with OS/390 V2R8.	
OS/390 V2R9	5647-A01	March 2000	OS/390 V2R9 C/C++ is the follow-on to OS/390 V2R6 C/C++. It includes the following new compiler options and suboptions: • `CHECKOUT(CAST)` • `COMPRESS` • `CVFT` • `DIGRAPH` (for C) • `IGNERRNO` • `INITAUTO` • `IPA(OBJONLY)` • `PHASEID` • `ROCONST` • `ROSTRING` • `STRICT` • `TARGET` suboptions enhancements It also includes the following #pragma directives: • `leaves` • `option_override` • `reachable` OS/390 V2R9 Language Environment comes with OS/390 V2R9.	
OS/390 V2R10	5647-A01	Sept 2000	OS/390 V2R10 C/C++ is the follow-on to OS/390 V2R9 C/C++. It includes the following new compiler options and suboptions: • `COMPACT` • `GOFF` • `IPA(LEVEL(2))` • `XPLINK` and enhancements to the following compiler options and suboptions: • `SPILL` • `TARGET` It also includes improvements to: • `#pragma option_override` • Packed decimal optimization in C OS/390 V2R10 Language Environment comes with OS/390 V2R10.	
z/OS V1R1	5694-A01	Mar 2001	z/OS V1R1 C/C++ is functionally equivalent to OS/390 V2R10 C/C++.	

Table 2. A History of Compilers and Libraries (continued)

Short Name	Product Number	GA Date	Description	Service Status
z/OS V1R2	5694-A01	Oct 2001	z/OS V1R2 C/C++ is fully compliant with ISO C++ 1998 Standard, with support for: • namespaces • new type bool and associated keywords bool, true, and false • new class member modifying keywords mutable and explicit • new casts • new template model • Run Time Type Identification (RTTI) • C++ Standard Library, including the Standard Template Library (STL). It also includes the following enhancements: • IBM Open Class® Library new level • Enhanced ASCII and Large File support in C++ Standard I/O Stream Library • `IPA` support for `XPLINK`	

Chapter 2. Common Questions about Migration

This chapter describes the kind of migration impacts that you may encounter, and the possible solutions.

Will Existing Language Environment Applications Run with z/OS Language Environment V1R2?

Yes, in nearly all situations, existing well-behaved Language Environment applications can be run with z/OS Language Environment without any modifications. A well-behaved application is one that relies on documented interfaces only.

For example, the *z/OS C/C++ Run-Time Library Reference* states that the `remove()` function returns a nonzero return code when a failure occurs. The following code fragments show the correct and incorrect ways to call the `remove()` function and to check the return code:

Incorrect method

```
if (remove("my.file") == -1) {
   call_err();
}
     .
     .
     .
```

Correct method

```
if (remove("my2.file") != 0) {
   call_err();
}
     .
     .
     .
```

The value of the return code from the `remove()` function changed in LE/370 R3. If an LE/370 R2 program was coded incorrectly, and checked for a specific value, as in the first code fragment, a source change is required when the code is migrated. This situation is common when an application relies upon undocumented interfaces. However, if the program was coded correctly, and it did not check for a specific nonzero return code, as in the second fragment, no source changes are required.

Will Existing C/370 Applications Work with z/OS Language Environment V1R2?

A C/370 application is created using the IBM C/370 Version 1 or Version 2 compiler and library, or the AD/Cycle C/370 V1R2 compiler with the `TARGET(COMPAT)` option and the C/370 V2R2 library. A well-behaved C/370 application, in most situations, works with z/OS Language Environment without any modifications.

Two common migration problems that you may encounter relate to interlanguage calls:

- You must relink applications that contain interlanguage calls between C/370 and Fortran before running them with z/OS Language Environment
- You can only run them with z/OS Language Environment after they are relinked. You cannot continue to run them with the C/370 library.

The same rules apply to applications that contain interlanguage calls between C/370 and COBOL, unless you relink them with the C/370 V2R1 or V2R2 library with the PTF for APAR PN74931 applied. This PTF replaces the C/370 V2 link-edit stubs so that they tolerate Language Environment. After your application is relinked using the modified C/370 V2 stubs, you can run the application with either the C/370 V2 run-time library or with Language Environment. Refer to “Executable Programs with Interlanguage Calls” on page 16 for more information about COBOL and Fortran interlanguage calls.

Though there are other migration items (described in the following chapters) that may affect your application, these are the most serious ones.

My Application Does Not Run — Now What?

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 a new design feature in the run-time library. Do the following:

1. Verify the concatenation order of your libraries.

   If you have a load module built with both C/370 library parts and z/OS Language Environment parts, ensure that you are not accidentally initializing your environment using the C-PL/I Common Library rather than z/OS Language Environment. The PDS with the low level qualifier SCEERUN (which belongs to z/OS Language Environment), must be concatenated ahead of the PDS with the low level qualifier SIBMLINK (which belongs to the C-PL/I Common Library).

   Refer to the section “Initialization Compatibility” on page 17 for more information.

2. Use environment variables to obtain the “Old Behavior”.

   Under z/OS Language Environment, you can use the `ENVAR` run-time option to specify the values of environment variables at execution time. With some environment variables, you can specify the “old behavior” for particular items. The following setting provides you with “old behavior” for the greatest number of items:

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

   The value assigned to `_EDC_COMPAT` is used as a bit mask. If you assign a value of 32767, the library uses “old behavior” for all of the general compatibility items currently defined by `_EDC_COMPAT`. For more information about `_EDC_COMPAT` and its possible values, refer to the *z/OS C/C++ Programming Guide*.

   If `_EDC_COMPAT` solves your migration problem, you can use it with the `ENVAR` run-time option, as shown above, or in a call to `setenv()` either in the CEEBINT High-Level Language exit or in your `main()` program. Using CEEBINT only requires you to relink your application, but adding a call to `setenv()` in the `main()` function requires a recompile and obviously a relink. See the *z/OS C/C++ Run-Time Library Reference*, and the *z/OS C/C++ Programming Guide* for more details about the `setenv()` function.

3. Relink your application.

   Relinking your application with z/OS Language Environment ensures that you did not link in any non-z/OS Language Environment interfaces. You must relink your C/370 application before running it with z/OS Language Environment, if your application:

   - Contains ILCs between C and Fortran, or between C and COBOL.

     Refer to “Executable Programs with Interlanguage Calls” on page 16 for more information.

- Is an SPC application that uses the library
- Contains calls to `ctest()`

4. Review the migration items documented in this book.

   If you find a migration item in this manual that you think may affect your application, use the workaround described in this book. If a relink or a setting of an environment variable is not suggested, you must change your source, and then recompile and relink your application.

5. Look for uninitialized storage.

   In some cases, applications will run with uninitialized storage, because the run-time library may inadvertently clear storage, or because the storage location referenced is set to zero.

   Use the `STORAGE` and `HEAP` run-time options to find uninitialized storage. We recommend `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.

   Note: Your program will run slower with these options specified. Do not use them for production, only development.

6. Look for undocumented interfaces.

   It is possible that your application has dependencies on undocumented interfaces. For example, you may have dependencies on library control blocks, specific errno values, or specific return values. Alter your code to use only documented interfaces, and then recompile and relink.

7. Contact your service representative.

   If you followed steps 1 to 6, but cannot run your existing load module under z/OS Language Environment, contact your System Programmer to verify 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. If this is not the case, ask your Service Representative to open a Problem Management Record (PMR) against the applicable IBM product. See the `APAR` member in data set `CBC.SCCNDOC` for information on how to open a PMR.

I Attempt to Recompile My Application and It Fails — Why?

Changes were made between versions and releases of compilers. Several changes were made between C/370 V1 and C/370 V2. In some cases, these changes were made to ensure compliance with language standards such as ISO. This book describes these changes, and the alterations you may need to make to your code.

The amount of memory required by the compiler sometimes changes from release to release. If you cannot recompile an application that you successfully compiled with a previous release of the compiler, try increasing the region size.

Part 2. From C/370 to z/OS V1R2 C/C++

This part discusses the implications of migrating applications that were created with one of the following compilers and one of the following libraries to the z/OS V1R2 C/C++ product.

Compilers:

- The IBM C/370 V1 compiler, 5688-040
- The IBM C/370 V2 compiler, 5688-187
- The AD/Cycle C/370 V1R2 compiler with the `TARGET(COMPAT)` compiler option, 5688-216

Libraries:

- The IBM C/370 V1 library, 5688-039, and C-PL/1 Common Library, 5688-082
- The IBM C/370 V2 library, 5688-188, and C-PL/1 Common Library, 5688-082

In this part, z/OS V1R2 may also be referred to as z/OS Language Environment, or Language Environment.

Chapter 3. Application Executable Program Compatibility

This chapter will help application programmers understand the compatibility considerations of application executable programs.

An executable program is the output of the prelink/link or bind process. For more information on the relationship between prelinking, linking, and binding, see the section about prelinking, linking, and binding in *z/OS C/C++ User's Guide*. The output of this process is a load module when stored in a PDS and a program object when stored in a PDSE or HFS.

Generally, C/370 executable programs execute successfully with z/OS Language Environment V1R2 without source code changes, recompilation, or relinking. This chapter highlights exceptions and shows how to solve specific problems in compatibility.

Executable program compatibility problems requiring source changes are discussed in "Chapter 4. Source Program Compatibility" on page 21.

Note: The terms in this section having to do with linking (bind, binding, link, link-edit) refer to the process of creating an executable program from object modules.

Input and Output Operations

Programs that ran with the C/370 V1 or V2R1 library may have to be changed to run with z/OS Language Environment if they have dependencies on any of the input and output behaviors listed in "Chapter 7. Input and Output Operations Compatibility" on page 41.

Differences Between the C/370 V1 and V2 Compilers

If you have programs that were created with C/370 V1, you should be aware of some changes made in C/370 V2 that may affect them. These differences also exist in the z/OS C compiler. See "Chapter 5. C/370 V1 to C/370 V2 Compiler Changes" on page 27 for more information.

Executable Programs That Invoke Debug Tool or dbx

When migrating your application from C/370 to z/OS Language Environment V1R2, you must relink modules that contain calls to `ctest()`. The old library object, @@CTEST, must be replaced as described in "Converting Old Executable Programs to New Executable Programs" on page 18 and in "Considerations for Interlanguage Call (ILC) Applications" on page 19. After you replace the old objects, the new modules are executable under z/OS Language Environment.

System Programming C Facility (SPC) Executable Programs

There are two types of SPC programs: the ones that still require the run-time library, and the ones that do not. With z/OS Language Environment, only the SPC executable programs that use the z/OS C/C++ run-time library need to be relinked. You can relink applications from executable programs or from text decks using the z/OS Language Environment text libraries. If you relink from text decks, you can use the JCL that originally built the application. However, you must modify it to point

to the z/OS Language Environment static or resident library (SCEELKED). If you relink from executable programs, you will need to do a CSECT replacement for the appropriate part, such as `EDCXSTRL`, `EDCXENVL`, and `EDCXHOTL`.

If your SPC module has been built with exception handling, automatic library call is not enabled when you relink, so you must explicitly include the new routine @@SMASK.

Executable Programs with Interlanguage Calls

You must relink C/370 executable programs that contain interlanguage calls (ILCs) to or from COBOL to execute them under z/OS Language Environment. Old executable programs that contain ILCs to and from assembler or PL/I language modules do not need to be relinked.

To relink your C/370-COBOL ILC application under the C/370 V2R2 library so that it can run under either the C/370 V2R2 library or Language Environment, obtain and apply PTF for APAR PN74931 for the V2R1 or V1R2 link-edit stubs. This PTF replaces the link-edit stubs so that they tolerate Language Environment. After your application is relinked using the modified V2, you can run the application with either the V2R1 or V2R2 run-time library, or with Language Environment.

To relink your C/370-COBOL ILC application so that it will only run under z/OS Language Environment, replace the old library objects @@C2CBL and @@CBL2C, as described in “Converting Old Executable Programs to New Executable Programs” on page 18 and “Considerations for Interlanguage Call (ILC) Applications” on page 19. After you replace the old objects, the new modules will be executable only under z/OS Language Environment.

Fortran-C ILC was not supported prior to Language Environment V1R5 and C/MVS V3R1, for Language Environment conforming applications. To use Fortran and C ILC routines, you must relink all Fortran-C ILC applications containing pre-Language Environment C or Fortran library routines.

The following table outlines when a relink of ILC applications is required, based on languages found in the executable program:

Table 3. Migrations requiring relinking

Language	Relink required
Assembler	No
PL/I	No
Fortran	YES
COBOL	YES *

Notes:

1. * 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 V1R2 C/C++. Otherwise a relink is required.
2. 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 Applications* for more information.

Initialization Compatibility

Both z/OS Language Environment V1R2 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 z/OS Language Environment V1R2 modules. The key static code for both C/370 and z/OS Language Environment modules is an object named CEESTART, which controls initialization at execution. Its contents differ between the products, thus there is an old and a new version of CEESTART. The key dynamic code for z/OS Language Environment is CEEBINIT, which is stored in SCEERUN. The key dynamic code for IBM C/370 Version 1 and Version 2 is IBMBLIIA, which is a Common Library part stored in SIBMLINK. The Common Library is used by the C/370 V1 and V2 libraries.

The following lists describe the initialization schemes:

IBM C/370 Version 1 and Version 2 Initialization

1. Old CEESTART loads IBMBLIIA.
2. IBMBLIIA initializes the Common Library.
3. The Common Library runs C/370-specific initialization.
4. The main program is called.

z/OS Language Environment Initialization

1. The new CEESTART loads CEEBINIT.
2. CEEBINIT initializes z/OS Language Environment.
3. z/OS Language Environment C-specific initialization is run.
4. The main program is called.

z/OS Language Environment Initialization of C/370 Executable Programs

1. Old CEESTART loads CEEBLIIA (as IBMBLIIA).
2. CEEBLIIA (IBMBLIIA) initializes z/OS Language Environment.
3. z/OS Language Environment C-specific initialization is run.
4. The main program is called.

In the third situation listed above, compatibility with old executable programs depends upon the program's ability to intercept the initialization sequence at the start of the dynamic code and to perform the z/OS Language Environment initialization at that point. This interception is done by providing a part named CEEBLIIA, assigned the alias of IBMBLIIA. This provides "initialization compatibility".

Special Considerations: CEEBLIIA and IBMBLIIA

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

To initialize the Common Library environment, ensure that SIBMLINK is concatenated before SCEERUN. To initialize the z/OS Language Environment environment, ensure that SCEERUN is concatenated before SIBMLINK. The version of IBMBLIIA that is found first determines the environment (Language Environment or Common Library) that is initialized.

Converting Old Executable Programs to New Executable Programs

Many sites will have some old executable programs that will require the C/370 Common Library environment unless they have been converted to use z/OS Language Environment. These are incompatible modules that, for example, contain ILCs to COBOL or that use the library function `ctest()` to invoke the Debug tool.

There are three different methods of converting old modules to new modules, so that they will run under z/OS Language Environment:

- Link from original objects using z/OS Language Environment. EDCSTART and CEEROOTB must be explicitly included.
- Relink the old executable program with z/OS Language Environment using CSECT replacement. EDCSTART and CEEROOTB must be explicitly included.

  Figure 1 shows an example of a job that uses this method. The job converts an old executable program to a new executable program by relinking it and explicitly including the z/OS Language Environment CEESTART to replace the old C/370 CEESTART.

  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 "Considerations for Interlanguage Call (ILC) Applications" on page 19 for the specific control statements that are necessary for different kinds of ILCs with COBOL.

```
//Jobcard information
//*
//*************************************************************************//
//*RELINK C/370 V1 or 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 1. Link Job for Converting Executable Programs

- For those modules that have a C `main()`, replace the old executable program by recompiling the source (if available). Recompile the source containing the `main()` function with the z/OS V1R2 C/C++ compiler, and then relink the objects with

z/OS Language Environment. This creates a version of CEESTART for z/OS Language Environment. This is an alternative to explicitly including EDCSTART when linking from objects.

Considerations for Interlanguage Call (ILC) Applications

This section lists the linkage editor control statements required to relink modules that contain ILCs between C and COBOL, and C and Fortran. The object modules are compatible with the z/OS Language Environment; however, the ILC linkage between the applications and the library has changed. You must relink these applications using the JCL shown in Figure 1 on page 18 and the control statements that fit your requirements from the following list. The `INCLUDE SYSLIB(@@CTDLI)` is only necessary if your program will invoke IMS™ facilities using the z/OS C library function `ctdli()` and if the z/OS 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*.

Chapter 4. Source Program Compatibility

This chapter describes the changes that you may have to make to your source code when moving applications to the z/OS V1R2 C/C++ product.

It considers programs created with one of the following compilers and one of the following libraries.

Compilers:

- The IBM C/370 V1 compiler, 5688-040
- The IBM C/370 V2 compiler, 5688-187
- The AD/Cycle C/370 V1R2 compiler with the `TARGET(COMPAT)` compiler option, 5688-216

Libraries:

- The IBM C/370 V1 library, 5688-039, and C-PL/1 Common Library, 5688-082
- The IBM C/370 V2 library, 5688-188, and C-PL/1 Common Library, 5688-082

C/370 V1 modules were created with the C/370 V1 library. C/370 V2 modules were created with the C/370 V2 library.

"Chapter 6. Other Migration Considerations" on page 31 has information on run-time options, which may also affect source code compatibility.

Input and Output Operations

You may have to change programs running with the C/370 V1 or V2R1 library if they have dependencies on any of the input and output behaviors listed in "Chapter 11. Input and Output Operations Compatibility" on page 65.

Differences Between the C/370 V1 and V2 Compilers

If you have programs that were created with the C/370 V1 compiler, you should be aware of some changes made in C/370 V2 that may affect your programs. These differences are also in the z/OS C compiler. See "Chapter 5. C/370 V1 to C/370 V2 Compiler Changes" on page 27 for more information.

SIGFPE Exceptions

Decimal overflow conditions were masked in the C/370 library before V2R2. The conditions were enabled when the packed decimal data type was introduced in the AD/Cycle C/370 V1R2 compiler, and continue to be enabled with z/OS Language Environment V1R2. If you have old load modules (created with the C/370 V1 or V2R1 library) that accidentally generated decimal overflow conditions, they may behave differently with z/OS Language Environment, by raising unexpected `SIGFPE` exceptions. You cannot migrate such modules to the new library without altering the source, and they are unsupported.

It is unlikely that such modules are present in a C-only environment. These unexpected exceptions may occur in mixed language modules, particularly those using C and assembler code where the assembler code explicitly manipulates the program mask.

Program Mask Manipulations

Programs created with the C/370 V1 or V2R1 compiler and library that explicitly manipulated the program mask may require source alteration to execute correctly under z/OS Language Environment. Changes are required if you have one of the following types of programs:

- A C program containing interlanguage calls (ILCs), where the invoked code uses the S/370™ decimal instructions that might generate an unmasked decimal overflow condition, requires modification for migration. There are two methods for migrating the code. The first one is preferred:
  - 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 signal is ignored, and then reenabled:

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

- 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.

The release() Function

With the z/OS C compiler and z/OS Language Environment, you can no longer issue a `release()` call against a fetched COBOL, Fortran, or PL/I module. If you do, `release()` returns a nonzero return code. You can still use `release()` with C modules and non-z/OS Language Environment enabled assembler modules.

If your application fetches and releases PL/I, Fortran, or COBOL modules, you must change your source code, then recompile and link it with z/OS Language Environment, if you are dependent on the `release()` return code.

Although `release()` could be issued against any assembler routines with IBM C/370 Version 1 and Version 2, it cannot be issued against z/OS Language Environment-enabled assembler routines. These routines, known as ASM15 routines, are assembled with z/OS Language Environment assembler prologs. ASM15 routines are coded with the CEEENTRY macro. If any assembler routines are rewritten as ASM15 routines, ensure that the calling code does not issue a `release()` call against them.

The realloc() Function

When the `realloc()` function is used with z/OS Language Environment, a new area is always obtained and the data is copied. This is different from IBM C/370 Version 1 and Version 2, where, if the new size was equal to or less than the original size, the same area was used.

Programmers may want to ensure that their source code has no dependencies on the behavior of the old version of the `realloc()` function, so that their code is compatible with z/OS Language Environment.

Fetched Main Programs

C/370 V1 and V2 programs that are fetched must now be recompiled without a main entry point. Under z/OS Language Environment, if you attempt to fetch a main program it will fail.

User Exits

If both CEEBXITA and IBMBXITA are present in a relinked IBM C/370 Version 1 or Version 2 module, CEEBXITA will have precedence over IBMBXITA.

#line Directive

The AD/Cycle C/370 V1R2 compiler ignored the `#line` directive when either the `EVENTS` or the `TEST` compiler option was in effect. The z/OS C compiler does not ignore the `#line` directive.

sizeof Operator

The behavior of `sizeof` when applied to a function return type was changed in the C/C++ MVS V3R2 compiler. For example:

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

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

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

With the OS/390 V2R4 C/C++ and subsequent compilers, you can use `#pragma wsizeof` or the `WSIZEOF` compiler option to get `sizeof` to return the widened size for function return types if your code has a dependency on this behavior. For more information on `#pragma wsizeof`, see *C/C++ Language Reference*. For more information on the `WSIZEOF` compiler option, see *z/OS C/C++ User's Guide*.

System Programming C Applications Built with EDCXSTRX

If you have SPC applications that are built with EDCXSTRX and that use dynamic C library functions, note that the name of the C library function module has changed from EDCXV in C/370 V2 to CEEEV003 in z/OS Language Environment. Change the name from EDCXV to CEEEV003 in the assembler source of your program that loads the library, and reassemble.

The __librel() Function

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. Under z/OS Language Environment, it has 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 the following table. The hexadecimal value is interpreted as 0xPVRRMMMM, where:

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

Product	librel value
C/370 V2R2	0x02020000
LE V1R5	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

In IBM C/370 V1 and V2, the high-order 8 bits were used to return the version number. Now these 8 bits are divided into 2 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 the *z/OS C/C++ Run-Time Library Reference*.

Library Messages

There are differences in messages between C/370 and z/OS Language Environment. Some run-time 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. **Do not build dependencies on message contents or message numbers.**

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

Prefix of perror() and strerror() Messages

All `perror()` and `strerror()` messages in C under z/OS Language Environment contain a prefix (in IBM C/370 Version 1 and Version 2 there were no prefixes to these messages). The prefix is EDC*xxxxa*, where *xxxx* is a number (always *5xxx*) and the *a* is either `I`, `E`, or `S`. See *z/OS Language Environment Run-Time Messages* for a list of these messages.

Compiler Messages and Return Codes

There are differences in messages and return codes between the C/370 compilers and the z/OS C compiler. Message contents have changed, and return codes for some messages have changed (errors have become warnings, and the other way around). Any application that is affected by message content or return codes must be updated accordingly. **Do not build dependencies on message content, message numbers, or return codes**. See *z/OS C/C++ Messages* for a list of messages.

_Packed Structures and Unions

With the z/OS 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 (or the other way around)

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

Alternate Code Points

The following alternate code points are not supported by the z/OS C compiler:

- 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 were supported by the C/370 and AD/Cycle C/370 compilers (the `NOLOCALE` option was required if you were using the AD/Cycle C/370 V1R2 compiler).

Chapter 5. C/370 V1 to C/370 V2 Compiler Changes

This chapter describes some of the changes made between the C/370 V1 and V2 compilers. These changes also appear in the z/OS C compiler. Read this section if you are migrating programs from C/370 V1.

Source Code Incompatibilities

This section describes the changes you may have to make to your source code when moving from C/370 V1.

Characters

You can no longer assign a `char` the value `''`. A character must be between the single quotation marks. Under C/370 V1, `''` was the same as `'\0'`.

A warning is now issued when the `CHECKOUT` compiler option is specified, and more than 4 bytes are assigned to a `char` or more than 2 bytes are assigned to a `wchar_t` constant. These restrictions did not apply under C/370 V1.

Sign extension now occurs when the `#pragma chars(signed)` directive is used. Thus the value of `'\xff'` is `-1` when `char`s are signed. When a `signed char` literal is converted to `int`, the sign extension will occur on the most significant specified byte that is not shifted out. These are changes from C/370 V1. See Table 4 for examples.

Table 4. Sign Extensions

Value of `signed char` literal	Value of `int`
`'\x80\x00'`	`(int)0xffff8000`
`'\x80\x00\x00'`	`(int)0xff800000`
`'\x80\x00\x00\x00'`	`(int)0x80000000`
`'\x80\x00\x00\x00\x80'`	`0x00000080`

Note: A hexadecimal escape sequence represents one char of data, so `'\x123456789'` is equivalent to `'\x89'`.

The #pragma comment Directive

If you are using the `#pragma comment` directive, you must now enclose the characters specified in double quotation marks. In C/370 V1, the double quotation marks were not required.

Structure Declarations

With the z/OS C compiler, you must declare a `struct` type before any function calls that contain the `struct` as one of its parameters. Otherwise, the `struct` in the function call will be incomplete and the parameter passed must be a pointer to `void`.

For example, the following program will not compile as desired because `struct st` in `func_call` is an incomplete `struct`. The call of `func_call` with a pointer to a `struct` will be an incompatible parameter type with the expected pointer to `void`.

```
int func_call (struct st *s); /* incomplete struct */

struct st { int x, y, z; };
```

From C/370 to z/OS V1R2

```
int main(void)
{
    struct st *t;
    func_call(t);             /* pointer to struct st but func_call  */
                             /* can only accept pointer to void     */
    printf("\n", t->y);
}
```

To solve this problem, add a declaration before the function declaration:

```
struct st;

int func_call (struct st *s);

struct st { int x, y, z; };
```

Note: If you have the following declaration,

```
extern struct S my_struct;
```

the type must be completed before the –> or `.` operators can be performed on `my_struct`. In C/370 V1, the `struct S` had to be a complete type at the time this declaration was reached.

Function Argument Compatibility

If you compile the following example under C/370 V1, the compiler fails to notice that the argument `func1` is incompatible with the prototype for `func2`. The `func2()` function requires a pointer to a `void` function with an argument of type `void *`, but an argument of type pointer to `void` function with an argument of type `int *` is passed instead. The z/OS C compiler will generate an error message in this situation.

```
void func2( void (*)(void *) );

void func1(int *);

main() {
    func2( func1 );
}
```

Pointer Considerations

According to the *ISO C Standard*, pointers to `void` types and pointers to functions are incompatible types. The C/370 V2, AD/Cycle C/370, C/MVS V3, and z/OS C compilers perform some type checking, such as in assignments, argument passing on function calls, and function return codes.

If you are not conforming to ISO rules for the use of pointer types, your run-time results may not be as expected, especially when you are using the compiler option `OPTIMIZE`.

With the C/370 V2, the AD/Cycle C/370, and the C/MVS V3 compilers, you could not assign `NULL` to an integer value. For example, the following was not allowed:

```
int i = NULL;
```

With the C/MVS V3R2 and z/OS C compilers, you can assign `NULL` pointers to void types if you specify `LANGLVL(COMMONC)` when you compile your program.

Macro Changes

In `stdio.h`, the `#define` macro `__VSAM_OPEN_AIX_PATH` (a value for the `__amrc` struct `__last_op` field), was replaced in C/370 V2 by `__VSAM_OPEN_ESDS_PATH` and `__VSAM_OPEN_KSDS_PATH`.

Modules compiled with C/370 V1 work with z/OS Language Environment. However, if you plan to compile your source with the z/OS C compiler, you must first change it to use `__VSAM_OPEN_ESDS_PATH` and `__VSAM_OPEN_KSDS_PATH`.

Chapter 6. Other Migration Considerations

This chapter provides additional considerations on migrating applications to z/OS V1R2 C/C++ that were created with one of the following compilers, and with one of the following libraries.

Compilers:

- The IBM C/370 V1 compiler, 5688-040
- The IBM C/370 V2 compiler, 5688-187
- The AD/Cycle C/370 V1R2 compiler with the `TARGET(COMPAT)` compiler option, 5688-216

Libraries:

- The IBM C/370 V1 library, 5688-039, and C-PL/1 Common Library, 5688-082
- The IBM C/370 V2 library, 5688-188, and C-PL/1 Common Library, 5688-082

Changes That Affect User JCL, CLISTs, and EXECs

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

Return Codes and Messages

Library return codes and messages have been changed, and JCL, CLISTs and EXECs that are affected by them must be changed accordingly (or else the CEEBXITA exit must be customized to emulate the old return codes). IBM C/370 Version 1 and Version 2 return codes were from 0 to 999. However, the z/OS Language Environment return codes have a different range. These return codes are documented in *z/OS Language Environment Debugging Guide*.

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.

Compiler message contents and return codes have changed. You must change JCL, CLISTs, and EXECs that are affected by them. Refer to “Compiler Messages and Return Codes” on page 25 for more information.

Changes in Data Set Names

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

Differences in Standard Streams

Under z/OS Language Environment there is no longer an automatic association of ddnames `SYSTERM`, `SYSERR`, `SYSPRINT` with `stderr`. Command line redirection of the type `1>&2` is necessary in batch to cause `stderr` and `stdout` to share a device.

In IBM C/370 Version 1 and Version 2, you could override the destination of error messages by redirecting `stderr`. z/OS Language Environment determines the destination of all messages from the `MSGFILE` run-time option. See the section on the `MSGFILE` run-time option in the *z/OS Language Environment Programming Guide* for more information.

Passing Command-Line Parameters to a Program

In IBM C/370 Version 1 or Version 2, 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 Language Environment, 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*.

SYSMSGS ddname

The method of specifying the language for compiler messages has changed. Instead of specifying a messages data set for the `SYSMSGS` ddname, you must now use the `NATLANG` run-time option. If you specify a data set for the `SYSMSGS` ddname, it will be ignored.

Run-Time Options

This section describes changes that may affect your run-time options.

Ending the Run-Time Options List

In C/370 V1 and V2, when passing only run-time options to a C/370 program, you did not have to end the arguments with a slash (/). With z/OS Language Environment, you must end the arguments with a slash.

With z/OS Language Environment, if you have no run-time options and the input arguments passed to `main()` contains a slash, you must prefix the arguments with a slash. JCL, CLISTs, and EXECs that are affected by the slash must be changed accordingly.

ISASIZE, ISAINC, STAE/SPIE, LANGUAGE, and REPORT options

Use the z/OS Language Environment equivalent for the IBM C/370 Version 1 and Version 2 run-time options on the command line and in `#pragma runopts`.

ISASIZE/ISAINC	becomes	`STACK`
LANGUAGE	becomes	`NATLANG`
REPORT	becomes	`RPTSTG`
SPIE/STAE	becomes	`TRAP`

The C/370 run-time options are mapped to z/OS Language Environment equivalents. However, if you do not use the z/OS 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.

STACK Default Size

The default size and increment for the `STACK` run-time option have changed. If you have not indicated the size and increment, `STACK` will be allocated differently when your program is running under z/OS Language Environment. The defaults in IBM C/370 Version 1 and Version 2 were 0K size and 0K increment. The defaults under

z/OS Language Environment without CICS® are 128K size, 128K increment, and BELOW, and with CICS are 4K size, 4080 increment, and ANYWHERE. With CICS the default location has changed to ANYWHERE.

To summarize, in z/OS Language Environment, the IBM-supplied defaults are STACK(128K,128K,BELOW,KEEP) without CICS and STACK(4K,4080,ANYWHERE,KEEP) with CICS.

STACK parameters

The parameters for the STACK run-time option are all positional in z/OS Language Environment; in IBM C/370 Version 1 and Version 2, only the first two were. The keyword parameter could be specified if the first two were omitted. Now, to specify only ANYWHERE you must enter: STACK(,,ANYWHERE).

HEAP Default Size

The default size and increment for the HEAP run-time option have changed. If you have not indicated the size and increment, HEAP will be allocated differently when running under z/OS Language Environment. The defaults in IBM C/370 Version 1 and Version 2 were 4K size and 4K increment. The defaults under z/OS Language Environment without CICS are 32K size and 32K increment and with CICS are 4K size and 4080 increment.

Two new parameters have been added, initsz24 and incrsz24. They determine how much of the heap is allocated and incremented below the 16M line.

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

To summarize, under z/OS Language Environment, the IBM-supplied defaults are HEAP(32K,32K,ANYWHERE,KEEP,8K,4K) without CICS and HEAP(4K,4080,ANYWHERE,KEEP,4K,4080) with CICS.

HEAP Parameters

In IBM C/370 Version 1 and Version 2, 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. Under z/OS Language Environment, all parameters are positional. To specify only KEEP, you must enter HEAP(,,,KEEP).

Compiler Options

This section describes changes that may affect your compiler options.

DECK compiler option

In IBM C/370 V1, the DECK compiler option directed the object module to the data set associated with SYSLIN. With the z/OS C compiler, as with the AD/Cycle C/370 and IBM C/370 V2 compilers, the object module is directed to the data set associated with SYSPUNCH.

As of z/OS V1R2, the DECK compiler option is no longer supported. The replacement for DECK functionality that routes output to DD:SYSPUNCH, is to use OBJECT(DD:SYSPUNCH).

HWOPTS compiler option

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

INLINE compiler option

The defaults for the INLINE compiler option have changed. In the past, the default for the threshold suboption was 250 ACUs (Abstract Code Units). With the C/MVS V3 and z/OS C compilers, the default is 100 ACUs.

OMVS compiler option

In IBM AD/Cycle V1, the OMVS compiler option directed the compiler to use the POSIX.2 standard rules for searching for files specified with #include directives. As of z/OS V1R2, the OMVS compiler option is no longer supported. The replacement for it is the OE option.

OPTIMIZE compiler option

In the C/370 V2R1 and subsequent compilers, OPTIMIZE mapped to OPT(1).

Starting with OS/390 V2R6, the C compiler maps both OPTIMIZE and OPT(1) to OPT(2).

SEARCH and LSEARCH compiler option

The include file search process has changed. Prior to the C/MVS V3R2 compiler, if you used the LSEARCH option more than once, the compiler would only search the libraries specified for the last LSEARCH option. Now the z/OS C compiler searches all of the libraries specified for all of the LSEARCH options, from the point of the last NOLSEARCH option.

Similarly, if you specify the SEARCH option more than once, the z/OS C 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.

TEST compiler option

Starting with 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) or OPT(2) is in effect.

Starting with C/C++ MVS V3R2, a restriction applies to the TEST compiler option if you are using the z/OS C/C++ compiler. Now, 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 z/OS Language Environment Dump Services are undefined.

Language Environment Run-Time Options

If occurrences of ISASIZE/ISAINC, STAE/SPIE, LANGUAGE, and REPORT runopts are specified by #pragma runopts in your source code, you may want to change them to the z/OS Language Environment equivalent before recompiling. These options are mapped to the z/OS Language Environment equivalent, but if you do not change them, you will get a warning or informational message during compilation.

Precedence of Language Environment over C/370 for #pragma runopts

If you link together C/370 and z/OS Language Environment object modules, and both modules contain `#pragma runopts`, the `#pragma runopts` settings in the Language Environment object module will take precedence.

System Programming C Facility Applications with #pragma runopts

If you code a program for use in the SPC environment and you use `#pragma runopts` to specify the heap or stack directives, the z/OS C compiler will expand these directives according to the z/OS Language Environment defaults and rules. Thus, the program may behave differently under z/OS Language Environment.

Decimal Exceptions

z/OS Language Environment provides support for the packed decimal overflow exception using native S/390® hardware enablement (as did the C/370 V2R2 library).

The value of the program mask in the program status word (PSW) is 4 (decimal overflow enabled).

Migration and Coexistence Considerations

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

- CICS programs running under z/OS Language Environment are enabled for decimal exceptions.
- The C packed decimal support routines are not supported in an environment that exploits asynchronous events.

SIGTERM, SIGINT, SIGUSR1, and SIGUSR2 Exceptions

There are changes to application/program behavior for `SIGTERM`, `SIGINT`, `SIGUSR1`, and `SIGUSR2` exceptions from C/370 V1 and V2.

The differences or incompatibilities are:

- The defaults for the `SIGINT`, `SIGTERM`, `SIGUSR1`, and `SIGUSR2` signals changed in LE/370 Release 3, from what they were in C/370 V1 and V2 and LE/370 R1 and R2. These changes were carried into z/OS Language Environment V1R2. In the C/370 library and LE/370 R1 and R2, the defaults for `SIGINT`, `SIGUSR1`, and `SIGUSR2` were to ignore the signals. As of LE/370 R3, the defaults are to terminate the program and return a return code of 3000. For `SIGTERM`, the default has always been to terminate the program, but the return code is now 3000 whereas before it was 0.
- Applications that terminate abnormally will **not** drive the `atexit` list.

Running Different Versions of the Libraries under CICS

You cannot run two different versions of the C/370 run-time libraries within one CICS region.

Sometimes a C/370 Version 2 CICS interface (EDCCICS) and the z/OS Language Environment CICS interface can be present in a CICS system through CEDA/PPT

definitions and inclusion of modules in the APF STEPLIB. Even if both versions are present, the z/OS Language Environment version will be initialized by CICS when the region is initialized.

CICS Abend Codes and Messages

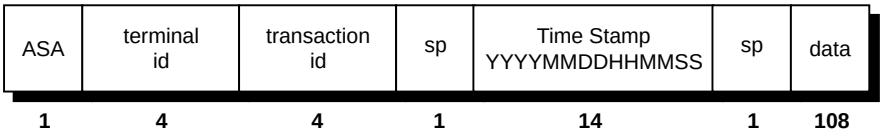
Abend codes such as ACC2 that were used by IBM C/370 Version 1 and Version 2 under CICS are not issued under z/OS Language Environment. An equivalent z/OS Language Environment abend code is issued instead; for example, 4nnn.

CICS Reason Codes

Reason codes that appeared in the CICS message console log have been changed. The new ones are documented in the *z/OS Language Environment Debugging Guide*.

Standard Stream Support under CICS

Under CICS, with z/OS Language Environment, records sent to the transient data queues associated with `stdout` and `stderr` with default settings take the form of a message as follows:



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`.

C/370 V1 and V2 used a different format.

stderr Output under CICS

Output from `stderr` is sent to the CICS transient data queue, CESE. CESE is also used by z/OS Language Environment for run-time error messages, dumps, and storage reports. If you previously used this file exclusively for C/370 `stderr` output, you should note that the output may be different.

Transient Data Queue Names under CICS

Transient data queue names are mapped as follows under z/OS Language Environment:

OLD NAME	NEW NAME
CCSI	CESI
CCSO	CESO
CCSE	CESE

HEAP Option Used with the Interface to CICS

In C/370 V1R2 and V2, the location of heap storage under CICS was primarily determined by the residence mode (RMODE) of the program.The logic for determining the location of heap was as follows:

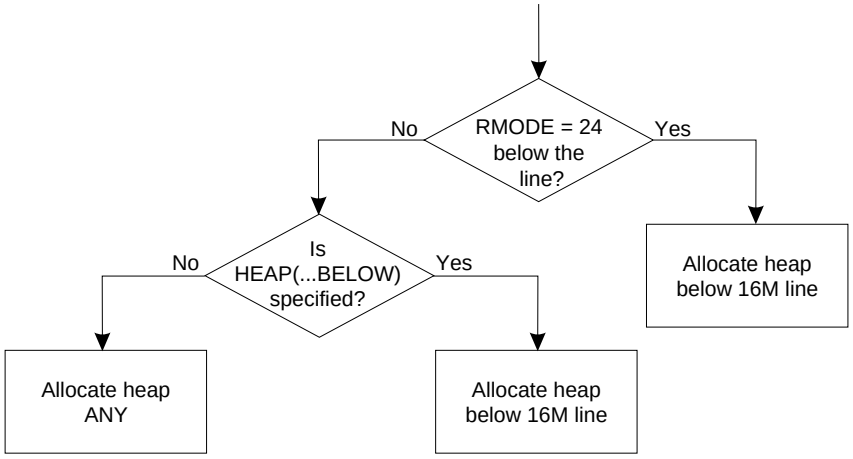


Figure 2. Heap Location Logic

With z/OS Language Environment, the location of heap storage is determined only by the `HEAP(...ANYWHERE|BELOW)` options. RMODE does not affect where the heap is allocated. Where the location of heap storage is important, you may want to change source accordingly.

COBOL Library Routines

All of the language libraries in z/OS Language Environment are packaged as a single unit in SCEERUN. Because of this packaging, for C-only applications, z/OS Language Environment V1R2 has the potential to invade the user's name space.

For example, z/OS Language Environment-enabled COBOL is available under z/OS Language Environment V1R2, and the following situations are possible:

- If there is a user C function name prefixed with IGZ or ILB that happens to correspond to a COBOL routine, there is the chance of binding in the COBOL routine at link-edit time.
- If there is a `fetch()` statement for a routine with a name prefixed with IGZ or ILB that happens to correspond to a COBOL routine that is dynamically loaded, it is possible that the COBOL routine will be loaded at run time.

To prevent the first problem, specify the user link libraries ahead of the z/OS Language Environment link libraries.

To prevent the second problem, specify the user execution libraries ahead of the z/OS Language Environment execution libraries.

Passing Control to the Cross System Product

As in IBM C/370 Version 1 and Version 2, control can be passed between Cross System Product (CSP) and z/OS Language Environment in three ways: `XFER`, `DXFR`, and `CALL`.

From C/370 to z/OS V1R2

If you have code that passes control from z/OS Language Environment V1R2 to CSP, which in turn passes control back to z/OS Language Environment V1R2, the behavior is undefined. Code that passes control from CSP to z/OS Language Environment V1R2, which in turn passes control back to CSP, is supported. In summary, z/OS Language Environment V1R2 must appear only once in the chain of passed control.

Syntax for the CC Command

With C/C++ MVS V3R2 and subsequent products, the CC command can be invoked using a new syntax. At customization time, your system programmer can customize the CC EXEC to accept only the old syntax (the one supported by compilers prior to C/MVS V3R2), only the new syntax, or both syntaxes.

You should customize the CC EXEC to accept only the new syntax, because the old syntax may not be supported in the future. If you customize the CC EXEC to accept only the old syntax, keep in mind that it does not support Hierarchical File System (HFS) files. 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. If you invoke this EXEC with the old syntax, it will not support HFS files.

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

atexit List during abort()

Unlike under IBM C/370 Version 1 and Version 2, the `atexit` list is not driven after a call to `abort()` under z/OS Language Environment.

Time Functions

For z/OS Language Environment, in the absence of customized locale information, the `ctime()`, `localtime()`, and `mktime()` functions return `Coordinated Universal Time (UTC)`.

If you were running with the C/370 V2R2 library, and applied both PTF UN61216 and PTF UN77602, or did not apply either one, the functions will return local time in the absence of customized locale information. Therefore, you will see a change in behavior beginning in OS/390 V2R6 Language Environment.

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

Direction of Compiler Messages to stderr

All messages generated by the z/OS C/C++ compiler are sent to `stderr`. In the past, some messages were sent to `stdout`.

Compiler Listings

As of OS/390 C/C++ V2R6, `OPT(1)` maps to `OPT(2)`. The compiler listing no longer conforms to the format 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 listings for the current release, refer to the *z/OS C/C++ User's Guide*.

Chapter 7. Input and Output Operations Compatibility

Changes were made to input and output support in the C/370 V2R2 and LE/370 V1R3 libraries. These changes also apply to z/OS Language Environment V1R2. If your programs performed input and output operations with the following products, you should read the changes listed in this section. These products are:

- The C/370 V1 library
- The C/370 V2R1 library

References in this chapter to previous releases or previous behavior apply to the products listed above.

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 only specified 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 Language Environment to ignore this parameter, as it did previously.

However, you may still need to change even well-behaved code under circumstances described in the following section.

Opening 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.

Writing to Files

- Write operations to files opened in binary mode are no longer deferred. Previously, the library did not write a block that held *nn* bytes out to the system until the user wrote *nn*+1 bytes to the block. The z/OS Language Environment library follows the rules for full buffering, described in *z/OS C/C++ Programming Guide*, and writes data as soon as the block is full. The *nn* 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 libraries 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.
- z/OS Language Environment blocks files more efficiently than some previous libraries 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 fclose(), fopen()
`abc\n\n\n`	`abc\n\n\n\n`
`abc\n\n`	`abc\n\n\n`
`abc\n`	`abc\n`

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

Written to file	Read from file after fclose(), fopen()
`abc\n\n\n`	`abc\n\n\n`
`abc\n\n`	`abc\n\n`
`abc\n`	`abc\n`

In previous products, writing a single new-line character to a new file created an empty file under MVS. z/OS Language Environment treats 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. The library writes a single blank to the file. When you read this file, you see two new-line characters instead of one. You also see two new-line characters on a read if you have written two new-line characters 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 5. 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

Table 5. Appending to ASA Files (continued)

abc Written to File, fclose() then append xyz	**What You Read from File after fclose(), fopen()**	
	Previous release	**New 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\rxyz\n	same as previous release

- The behavior of DBCS strings has changed.
  1. I/O now checks the value of `MB_CUR_MAX` to determine whether to interpret DBCS characters within a file.
  2. 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.
  3. 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.
  4. 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.
  5. 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`.

Repositioning within Files

- 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 new _EDC_COMPAT environment variable so thatsource code need not change to compensate for the new behavior. _EDC_COMPAT is described in *z/OS C/C++ Programming Guide*.
- 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. _EDC_COMPAT is described in *z/OS C/C++ Programming Guide*.
- 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. _EDC_COMPAT is described in *z/OS C/C++ Programming Guide*.
- If you have used ungetc() to move the file pointer to a position before the beginning of the file, calls to ftell() and fgetpos() now fail. Previously, ftell() returned the value 0 for such calls, but set errno to a non-zero value. Previously, fgetpos() did not account for ungetc() calls. See *z/OS C/C++ Programming Guide* for information on how to change fgetpos() behavior by using _EDC_COMPAT.

  For example, suppose that you are at relative position 1 in the file and ungetc() is performed twice. ftell() and fgetpos() will now report the relative position -1, which is before the start of the file, causing both ftell() and fgetpos() to fail.
- After you have called ftell(), calls to setbuf() or setvbuf() may fail. Applications should never call I/O functions between calls to fopen() or freopen() and calls to the functions that control buffering.

Closing and Reopening ASA Files

The behavior of ASA files when you close and reopen them is now consistent:

Table 6. Closing and Reopening ASA Files

	Physical record after close			
Written to file	**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)	

Table 6. Closing and Reopening ASA Files (continued)

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

fldata() Return Values

There are minor changes to the values that the `fldata()` library function returns. It may now return more specific information in some fields. For more information on `fldata()`, see the “Input and Output” section in *z/OS C/C++ Programming Guide*.

Error Handling

The general return code for errors is now `EOF`. In previous products, some I/O functions returned 1 as an error code to indicate failure. This caused some confusion, as 1 is a possible `errno` value as well as a return code. `EOF` is not a valid `errno` value.

Programs that rely on specific values of `errno` may not run as expected, because certain `errno` values have changed. Starting with OS/390 Language Environment V1R5, error messages have the format EDC5xxx. You can find the error message information for a particular `errno` value by applying the `errno` value to EDC5xxx (for example, 021 becomes EDC5021), and looking up the EDC5xxx message in *z/OS Language Environment Debugging Guide* manual.

Miscellaneous

- The inheritance model for standard streams now 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
`L_tmpnam`	47	1024
`FILENAME_MAX`	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.

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.

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.
- 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.
- 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.

- 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. From Pre-OS/390 Releases of C/C++ to z/OS V1R2 C/C++

This part discusses the implications of migrating applications that were created with one of the following compilers and one of the following libraries to the z/OS V1R2 C/C++ product.

Compilers:

- The AD/Cycle C/370 V1R1 compiler, 5688-216
- The AD/Cycle C/370 V1R2 compiler without the `TARGET(COMPAT)` compiler option (refer to Part 2 when the `TARGET(COMPAT)` option is specified), 5688-216
- The IBM C/C++ for MVS/ESA V3R1 compiler, 5655-121
- The IBM C/C++ for MVS/ESA V3R2 compiler, 5655-121
- IBM OS/390 C/C++ V1R1 compiler, 5645-001

Libraries:

- IBM SAA AD/Cycle Language Environment/370 V1R1, 5688-198
- IBM SAA AD/Cycle Language Environment/370 V1R2, 5688-198
- IBM SAA AD/Cycle Language Environment/370 V1R3, 5688-198
- Language Environment V1R4, 5688-198
- Language Environment V1R5, 5688-198
- The OpenEdition AD/Cycle C/370 Language Support Feature of MVS/ESA SP V5R1, 5655-068 and 5655-069
- The C/C++ Language Feature of MVS/ESA SP V5R2, 5655-068 and 5655-069
- The C/C++ Language Feature of MVS/ESA SP V5R2, 5655-068 and 5655-069
- OS/390 V1R1 Language Environment, 5645-001

Notes:

1. The OS/390 V1R1 compiler and library were equivalent to the final MVS/ESA compiler and library.
2. The z/OS V1R2 C++ compiler supports the ISO 1998 standard. There may be some changes in compiler behavior that are incompatible with previous C++ compilers. Refer to “Chapter 15. OS/390 V2R10 to z/OS V1R2 C/C++ Compiler Changes” on page 93 for more details.

This part does not discuss converting a C program to C++. The only C++ compiler migration considerations covered are those between different versions of the C++ component of the IBM C/C++ for MVS/ESA compilers and the z/OS V1R2 C/C++ compiler.

In this part, references to the products in the first column of the following table also apply to the products in the second column.

References To These Products	Also Apply To These Products
LE/370 R3	MVS/ESA SP V5R1 OpenEdition AD/Cycle C/370 Language Support Feature
Language Environment R4	C/C++ Language Feature of MVS/ESA SP V5R2
Language Environment R5	C/C++ Language Feature of MVS/ESA SP V5R2 (Modification 2)

References To These Products	Also Apply To These Products
OS/390 R1	IBM C/C++ for MVS V3R2 compiler and Language Environment R5

Chapter 8. Application Executable Program Compatibility

This chapter will help application programmers understand the compatibility considerations of application executable programs.

An executable program is the output of the prelink/link or bind process. For more information on the relationship between prelinking, linking, and binding, see the section about prelinking, linking, and binding in *z/OS C/C++ User's Guide*. The output of this process is a load module when stored in a PDS and a program object when stored in a PDSE or HFS.

Generally, C/370 executable programs execute successfully with z/OS Language Environment V1R2 without source code changes, recompilation, or relinking. This chapter highlights exceptions and shows how to solve specific problems in compatibility.

Executable program compatibility problems requiring source changes are discussed in "Chapter 9. Source Program Compatibility" on page 53.

Note: The terms in this section having to do with linking (bind, binding, link, link-edit) refer to the process of creating an executable program from object modules.

Input and Output Operations

Programs running with LE/370 V1R1 or V1R2 may not work without changes if they have dependencies on any of the input and output behavior listed in "Chapter 11. Input and Output Operations Compatibility" on page 65.

System Programming C Facility (SPC) Executable Programs

If you have an LE/370 V1R1 or V1R2 SPC application that was built with exception handling (that is linked with EDCXERR, EDCXABRT and EDCXHDLR), you must relink it with z/OS Language Environment V1R2 using the SCEESPC data set.

If your SPC module was built with exception handling, automatic library call is not enabled when you relink, so you must explicitly include the new routine @@SMASK.

Using the LINK Macro to Initiate a main()

When the LINK macro was used to initiate one C `main()` from another in LE/370 V1R0, any run-time options specified in calling a child `main()` were ignored. The parent run-time options were inherited. The conditions left unhandled in the child were propagated to the parent. Starting with LE/370 V1R1, and continuing through to z/OS Language Environment V1R2 run-time options are no longer propagated.

With LE/370 V1R0, using LINK to initiate a child `main()` restricted you from using standard streams in the child and from using memory files in the child. Starting with LE/370 V1R1 and continuing through to z/OS Language Environment V1R2, these restrictions no longer apply. Therefore, the parent's standard streams and memory files are shared by the child.

Inheritance of Run-Time Options with EXEC CICS LINK

When an EXEC CICS LINK command was used with LE/370 V1R1, run-time options were inherited from an ancestor. Users who used STACK and HEAP to tune C-CICS applications had to take particular note of this. Because of this inheritance, a large heap or stack size specified in the first run unit of a transaction chain of run units could cause shortages when it was allocated for each unit. For programs running under later releases of Language Environment, including z/OS Language Environment V1R2, run-time options are no longer inherited.

STAE/NOSPIE and SPIE/NOSTAE Mapping

STAE and SPIE options have been replaced with the TRAP option. We recommend that you use the TRAP 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. If the STAE option and SPIE option are used, they map to TRAP(ON,SPIE). If NOSTAE and NOSPIE are used, they map to TRAP(OFF). When the values are mixed, for example, STAE/NOSPIE, they map to TRAP(ON,SPIE). In LE/370 V1R1, SPIE/NOSTAE and STAE/NOSPIE are mapped to TRAP(OFF).

Class Library Execution Incompatibilities

There are execution incompatibilities between the class libraries provided with the IBM C/C++ for MVS/ESA V3R1M0, V3R1M1 and V3R2M0. You must recompile and relink applications that are dynamically bound to those class libraries for the following migration paths:

- Collection Class
  - From C++ for MVS/ESA V3R1M0, V3R1M1 or V3R2M0 to z/OS V1R2 C/C++
- Application Support Class
  - From C++ for MVS/ESA V3R1M0, V3R1M1 or V3R2M0 to z/OS V1R2 C/C++

As of z/OS V1R2, the IBM Open Class Library DLLs are built with XPLINK. All the DLLs from the previous versions/releases are built with NOXPLINK, including the OS/390 V2R10 DLLs which are available in z/OS V1R2. If you migrate to the z/OS V1R2 IBM Open Class Library DLLs and your application is not built XPLINK then you must specify the XPLINK(ON) run-time option, which may cause a performance degradation to the application. If you are not using XPLINK, then you should continue to use the OS/390 V2R10 level of the IBM Open Class Library DLLs by either using the OS/390 V2R10 C++ compiler or the z/OS V1R2 C++ compiler with the TARGET(OSV2R10) compiler option specified. Alternatively, you can use the static version of the z/OS V1R2 IBM Open Class Library by linking in the static library CBC.SCLBCPP2 which contains both XPLINK and NOXPLINK objects.

Refer to “Appendix. Class Library Migration Considerations” on page 109, “Collection Class Library Source Code Incompatibilities” on page 55, and “Class Library Object Module Incompatibilities” on page 57 for some background information about class libraries and compatibility considerations.

Chapter 9. Source Program Compatibility

In general, you can use source programs with the z/OS V1R2 C/C++ product without modification, if they were created with one of the following:

- AD/Cycle C/370 compiler running with Language Environment V1R2 or later
- C/MVS V3R1 or V3R2 compiler running with Language Environment V1R4 or later
- C++/MVS V3R1, and C++/MVS V3R2 programs running with Language Environment V1R4 or later

This chapter highlights the exceptions and shows how to solve specific problems in compatibility.

"Chapter 10. Other Migration Considerations" on page 57 has information on run-time options, which may also affect source code compatibility.

Input and Output Operations

You may have to change programs that ran with the LE/370 R1 or R2 library so that they work with z/OS Language Environment, if they have dependencies on any of the input and output behaviors listed in "Chapter 11. Input and Output Operations Compatibility" on page 65.

SIGFPE Exceptions

Decimal overflow conditions were masked in R1 and R2 of LE/370. These conditions were enabled when the packed decimal data type was introduced in the AD/Cycle C/370 R2 compiler, and continue to be enabled with z/OS Language Environment V1R2.

If you have old load modules that accidentally generated decimal overflow conditions, they may behave differently with z/OS Language Environment V1R2 by raising unexpected `SIGFPE` exceptions. Without source alteration, such modules cannot be migrated to the new library, and are unsupported. It is unlikely that such modules will occur in a C-only environment. These unexpected exceptions may occur in mixed language modules, particularly those using C and assembler code where the assembler code explicitly manipulates the program mask.

Program Mask Manipulations

Programs created with LE/370 R1 or R2 that explicitly manipulated the program mask may require source alteration to execute correctly under z/OS Language Environment V1R2. Changes are required if you have one of the following types of programs:

- A C program containing assembler interlanguage calls (ILC), in which the invoked code uses the S/370 decimal instructions that might generate an unmasked decimal overflow condition, requires modification for migration. There are two methods for migrating the code. The first one is preferred:
  - Modify the assembler code to 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 signal is ignored, and then reenabled:

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

- 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 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.

#line Directive

The AD/Cycle C/370 and C/MVS V3R1 compilers ignored the #line directive when either the EVENTS or the TEST compiler option was in effect. As of C/MVS V3R2, the compiler does not ignore the #line directive.

sizeof Operator

The behavior of sizeof when applied to a function return type was changed in the C/C++ MVS V3R2 compiler. For example:

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

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

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

If you are using OS/390 V2R4 C/C++ and subsequent compilers, and you require sizeof to return the widened size for function return types, then you must use #pragma wsizeof together with the WSIZEOF compiler option. For more information on #pragma wsizeof, see the *C/C++ Language Reference*. For more information on the WSIZEOF compiler option, see the *z/OS C/C++ User's Guide*.

_Packed Structures and Unions

If you are migrating from an AD/Cycle C/370 compiler to the z/OS 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 (or the other way around)

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

Alternate Code Points

The following alternate code points are not supported by the z/OS C/C++ compilers:

- 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 were supported by the C/370 and AD/Cycle C/370 compilers (the `NOLOCALE` option was required if you were using the AD/Cycle C/370 V1R2 compiler).

Supporting the ISO standard

As of z/OS V1R2 the C/C++ compiler supports the ISO 1998 standard. See "Chapter 15. OS/390 V2R10 to z/OS V1R2 C/C++ Compiler Changes" on page 93 for details. Note that the OS/390 V2R10 compiler and z/OS V1R2 compiler support different language levels; hence, they are both shipped as part of z/OS V1R2.

LANGLVL(ANSI)

Starting with the C/C++ MVS V3R2 compiler, if you specify `LANGLVL(ANSI)`, the compiler recognizes `char`, `unsigned char`, and `signed char` as three distinct types.

As of z/OS V1R2 C++, if you specify `LANGLVL(ANSI)`, the compiler will conform to the ISO C++ 1998 standard. See "Chapter 15. OS/390 V2R10 to z/OS V1R2 C/C++ Compiler Changes" on page 93 for details.

Compiler Messages and Return Codes

There are differences in messages and return codes between different versions of the compiler. Message contents have changed, and return codes for some messages have changed (some errors have become warning, and in very rare situations, some warnings have become errors). You must update accordingly any application that is affected by message contents or return codes. **Do not build dependencies on message content, message numbers, or return codes**. See *z/OS C/C++ Messages* for a list of compiler messages.

Collection Class Library Source Code Incompatibilities

There are source code incompatibilities between the native Collection Class Libraries available with the C++ for MVS/ESA V3R1 and z/OS C++ compilers. You must change your source code if you are migrating to z/OS C++ from C++ for MVS/ESA V3R1 and your application makes use of either of the following:

newCursor method
: The return type of the newCursor method is now a pointer to the abstract cursor class ICursor (*ICursor).

Deriving from Reference Classes
: Deriving from Reference Classes without overriding existing Collection Class member functions is still possible. However, you can no longer override existing Collection Class functions and use your derived Collection

Class in a polymorphic way without additional effort. Refer to the chapter about "Polymorphism and the Collections", in the *IBM Open Class Library User's Guide* for more information.

These changes were made in the Collection Class Library that was available with the C++ for MVS/ESA V3R1M1 class libraries, and thus only affect you if you are migrating from the C++ for MVS/ESA V3R1M0 class library.

Also, the structure of the Collection Classes changed in C++ for MVS/ESA V3R1M1. All classes, including the concrete classes, are now related in an abstract hierarchy. The abstract hierarchy makes use of virtual inheritance. When you subclass from a Collection Class and implement your own copy constructor, you must initialize the virtual base class IACollection<Element> in your derived classes. Therefore, if you subclassed from a concrete Collection Class that was shipped with C++ for MVS/ESA V3R1M0, and are migrating to the Collection Classes that are shipped with z/OS V1R2 C/C++, you will have to change the implementation of your copy constructor by adding the virtual base class initialization. Refer to the chapter about "Collection Class Changes", in the *IBM Open Class Library User's Guide* for more information.

Also refer to “Appendix. Class Library Migration Considerations” on page 109, “Class Library Execution Incompatibilities” on page 52, and “Class Library Object Module Incompatibilities” on page 57 for some background information about class libraries and compatibility considerations.

DSECT Utility

Header files generated by the DSECT utility now use `#pragma pack` rather than `_Packed` for packed structures. In rare cases, you may have to modify and recompile your code.

Chapter 10. Other Migration Considerations

This chapter provides additional considerations on migrating applications from the compilers and libraries listed in “Part 3. From Pre-OS/390 Releases of C/C++ to z/OS V1R2 C/C++” on page 49 to the z/OS V1R2 C/C++ feature.

Class Library Object Module Incompatibilities

There are object incompatibilities between the class libraries provided with the IBM C/C++ for MVS/ESA V3R1M0, V3R1M1 and V3R2M0 compilers and the libraries provided with the z/OS C++ compilers. You must recompile and relink applications that are dynamically bound to those class libraries, for the following migration paths:

- Collection Class
  - From C++ for MVS/ESA V3R1M0 to z/OS V1R2 C/C++
- Application Support Class
  - From C++ for MVS/ESA V3R1M0, V3R1M1 or V3R2M0 to z/OS V1R2 C/C++

Refer to “Appendix. Class Library Migration Considerations” on page 109, “Class Library Execution Incompatibilities” on page 52, and “Collection Class Library Source Code Incompatibilities” on page 55 for some background information about class libraries and compatibility considerations.

Removal of Database Access Class Library Utility

Starting with OS/390 V2R4 C/C++, the Database Access Class Library utility is no longer available.

Changes That Affect User JCL, CLISTs, and EXECs

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

CXX Parameter in JCL Procedures

With C++ for MVS/ESA V3R2, OS/390®, and z/OS C++ compilers, the CBCC, CBCCL, and CBCCLG procedures, which compile C++ code, now include parameter CXX. You must include this parameter if you have written your own JCL to compile a C++ program. Otherwise, you invoke the C compiler.

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

```
run-time options/CXX compiler options
```

Specifying Class Library Header Files at Compile Time

In OS/390 V2R10 and earlier compilers, using the following JCL on the compile step would work, although not recommended:

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

As of z/OS V1R2, the record size for the SCLBH data sets have been increased from LRECL=80 to LRECL=120. Due to this change, the SYSLIB shown above will no longer work, and must be removed from your JCL. The replacement for this is the `SEARCH` compiler option, as in the following example:

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

From Pre-OS/390 Releases to z/OS V1R2

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

SYSMSGS and SYSXMSGS ddnames

With the C/C++ for MVS/ESA V3R2 and z/OS 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 SYSMSGS and SYSXMSGS ddnames, you must now use the NATLANG run-time option. If you specify data sets for these ddnames, they are ignored.

Changes in Data Set Names

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

Decimal Exceptions

z/OS Language Environment provides support for the packed decimal overflow exception using native S/390 hardware enablement, as did LE/370 V1R3, Language Environment V1R4, and Language Environment V1R5.

The value of the program mask in the program status word (PSW) is 4 (decimal overflow enabled).

Migration and Coexistence

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

- As of LE/370 V1R3, CICS programs were enabled for decimal exceptions.
- The C packed decimal support routines are not supported in an environment that exploits asynchronous events.

SIGTERM, SIGINT, SIGUSR1, and SIGUSR2 Exceptions

As of LE/370 V1R3, there were changes to application/program behavior for SIGTERM, SIGINT, SIGUSR1, and SIGUSR2 exceptions from previous releases of the LE/370 product. These changes in behavior carried over into the z/OS Language Environment V1R2 product.

The differences or incompatibilities are:

- The defaults for the SIGINT, SIGTERM, SIGUSR1, and SIGUSR2 signals changed in LE/370 R3, from what they were in C/370 V1R1 and V1R2 and LE/370 V1R1 and V1R2. In the C/370 library and LE/370 V1R1 and V1R2, the defaults for SIGINT, SIGUSR1, and SIGUSR2 were to ignore the signals. As of LE/370 V1R3, the default is to terminate the program and return a return code of 3000. For SIGTERM, the default has always been to terminate the program, but the return code is now 3000 whereas before it was 0.
- Applications that terminate abnormally will **not** drive the atexit list.

Compiler Options

This section describes changes that may affect your compiler options.

DECK Compiler Option

In IBM C/370 V1, the DECK compiler option directed the object module to the data set associated with SYSLIN. With the OS/390 C and the z/OS V1R1 C compiler, as with the AD/Cycle C/370 and IBM C/370 V2 compilers, the object module is directed to the data set associated with SYSPUNCH.

As of z/OS V1R2, the DECK compiler option is no longer supported. The replacement for DECK functionality that routes output to DD:SYSPUNCH, is to use OBJECT(DD:SYSPUNCH).

ENUM Compiler Option

z/OS V1R2 introduces the ENUM option as a means for controlling the size of enumeration types. The default setting, ENUM(SMALL), provides the same behavior in previous releases of the compiler. If you want to use the ENUM option, it is recommended that the same setting be used for the whole application; otherwise, you may 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).

HALT Compiler Option

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

HWOPTS Compiler Option

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

INFO Compiler Option

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

INLINE Compiler Option

For C, the default for the INLINE compiler option was changed to 100 ACUs (Abstract Code Units) in the MVS/ESA V3R1 compiler. Hence, with the C for MVS/ESA V3 and the z/OS 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(COMPAT) Compiler Option

In IBM C++ for MVS/ESA V3, the LANGLVL(COMPAT) option directed the compiler to generate code that is compatible with older levels of C++. As of z/OS V1R2, the LANGLVL(COMPAT) compiler option is no longer supported.

OMVS Compiler Option

In IBM AD/Cycle V1, the OMVS compiler option directed the compiler to use the POSIX.2 standard rules for searching for files specified with #include directives. As of z/OS V1R2, the OMVS compiler option is no longer supported. The replacement for it is the OE option.

OPTIMIZE Compiler Option

In the AD/Cycle C/370 compilers:

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

In the C/C++ for MVS/ESA V3 compilers, and the OS/390 V1R1 compiler:

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

Starting with the OS/390 V2R6 C/C++ compiler:

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

While the OPT level mapping for the C/C++ for MVS/ESA V3 and OS/390 V2R6 compilers is the same, the optimization is different. The underlying compiler technology within these compilers has changed significantly.

SEARCH and LSEARCH Compiler Option

The include file search process has changed. Prior to the C for MVS/ESA V3R2 compiler, if you used the LSEARCH option more than once, the compiler searched only the libraries specified for the last LSEARCH option. Now the z/OS C compilers search all of the libraries specified for all of the LSEARCH options, from the point of the last NOLSEARCH option.

Similarly, if you specify the z/OS C/C++ SEARCH option more than once, the z/OS C++ compilers search 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.

SRCMSG Compiler Option

In IBM C++ for MVS/ESA V3, the SRCMSG option directed the compiler to add the corresponding source code lines to the diagnostic messages that are written to stderr. As of z/OS V1R2, the SRCMSG compiler option is no longer supported.

SYSLIB, USERLIB, SYSPATH and USERPATH Compiler Options

In IBM C++ for MVS/ESA V3 and IBM C/C++ for MVS/ESA V3R2, the SYSLIB, USERLIB, SYSPATH and USERPATH compiler options directed the compiler on where to find include files. As of z/OS V1R2, these compiler options are no longer supported. The replacement for them are the SEARCH and LSEARCH options.

TEST Compiler Option

Starting with 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) or OPT(2) is in effect.

Starting with C/C++ MVS V3R2, a restriction applies to the TEST compiler option. Now, 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 z/OS Language Environment Dump Services are undefined.

Length of External Variable Names

The z/OS V1R2 binder imposes a limit of 1024 characters for the length of external names. Both the OS/390 C++ compiler and z/OS C++ compiler may sometimes generate mangled names that are longer than this limit. This could occur more often when using the Standard Template Library with the z/OS V1R2 C++ compiler. Should you encounter this problem:

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

In a future release of the Program Management Binder, IBM will raise the limit to match that of the prelinker (32K-1 characters).

Syntax for the CC Command

With the C/C++ for MVS/ESA V3R2 and z/OS C/C++ compilers, the CC command can be invoked using a new syntax. 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), only the new syntax, or both syntaxes.

You should customize the CC EXEC to accept only the new syntax, because the old syntax may not be supported in the future. If you customize the CC EXEC to accept only the old syntax, keep in mind that it does not support Hierarchical File System (HFS) files. 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. If you invoke this EXEC with the old syntax, it does not support HFS files.

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

Time Functions

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

Abnormal Termination Exits

The abnormal termination exits CEEBDATX (for batch) and CEECDATX (for CICS) are now automatically linked at install time for z/OS Language Environment the sample exit is no longer required. These exits were only available in the sample library in LE/370 V1R3. They allow you to automatically produce a system dump (with abend code 4039), when abnormal termination occurs. In previous releases of Language Environment, only an LE formatted dump was generated (which continues to be produced under z/OS Language Environment V1R2).

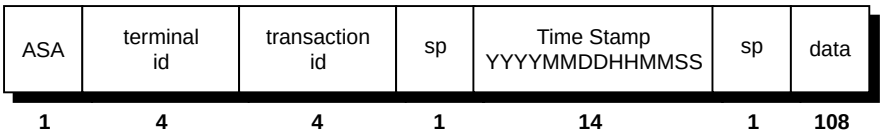
For a non-CICS application, you can trigger the dump by ensuring that SYSUDUMP is defined in the GO step of the JCL that you are using (for example, by including the statement SYSUDUMP DD SYSOUT=*). If SYSUDUMP is not included in your JCL, or is defined as DUMMY, the dump will be suppressed. As of C/C++ for MVS/ESA V3R1, the standard JCL procedures shipped with the compiler do not include SYSUDUMP.

In a CICS environment, you automatically receive the default transaction dump unless you disable it by using the CEMT transaction, and by specifying the dumpcode '4039'.

You may also modify CEEBDATX and CEECDATX to suppress the dumps. The exits are available in the z/OS Language Environment V1R2 sample library.

Standard Stream Support

Under CICS, with z/OS Language Environment, records sent to the transient data queues associated with `stdout` and `stderr` with default settings take the form of a message as follows:



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`

This format was associated with `stderr` for all releases of Language Environment. However, it has only been used for `stdout` since LE/370 Release 3; therefore, you should be aware of this change if you are migrating to z/OS Language Environment V1R2 from LE/370 V1R1 or V1R2.

Direction of Compiler Messages to stderr

All messages produced by the C/C++ for MVS/ESA V3R2 and z/OS C++ compilers are sent to `stderr`. In the past, some messages were sent to `stdout`.

Array new

In the C++ for MVS/ESA V3R1 compiler, the array version of `new` was not initially supported. It is supported in a PTF (APAR PN72107) available for the C++ for MVS/ESA V3R1 compiler, and it is also supported in the C++/MVS V3R2 and later C++ compilers.

If you are migrating from the base C/C++ for MVS/ESA V3R1 compiler to z/OS V1R2 C/C++, and you have written your own global `new` operator, it is no longer called when you create an array object. For example:

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

main() {
    X new_array[10]; // the global new operator
                     // shown above will not be called if the fix for
                     // APAR PN72107 or the V3R2
                     // compiler is installed
}
```

You have to add an overloaded operator to `new[]` if you require this for arrays.

Compiler Listings

As of OS/390 C/C++ V2 R6, `OPT(1)` maps 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 listings for the current release, refer to the *z/OS C/C++ User's Guide*.

Chapter 11. Input and Output Operations Compatibility

Changes were made to input and output support in the C/370 V2R2 and LE/370 Release 3 libraries. These changes also apply to z/OS Language Environment V1R2. If your programs performed input and output operations with the following products, you should read the changes listed in this section. These products are:

- LE/370 V1R1
- LE/370 V1R2

References in this chapter to previous releases or previous behavior apply to the products listed above.

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 only specified 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 Language Environment to ignore this parameter, as it did previously.

However, you may still need to change even well-behaved code under circumstances described in the following section.

Opening 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 a PDS or a PDSE, you cannot specify any spaces before the member name.

Writing to Files

- Write operations to files opened in binary mode are no longer deferred. Previously, the library did not write a block that held *nn* bytes out to the system until the user wrote *nn*+1 bytes to the block. The z/OS Language Environment library follows the rules for full buffering, described in *z/OS C/C++ Programming Guide*, and writes data as soon as the block is full. The *nn* 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 libraries 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.
- z/OS Language Environment blocks files more efficiently than some previous libraries 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 fclose(), fopen()
`abc\n\n\n`	`abc\n\n\n\n`
`abc\n\n`	`abc\n\n\n`
`abc\n`	`abc\n`

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

Written to file	Read from file after fclose(), fopen()
`abc\n\n\n`	`abc\n\n\n`
`abc\n\n`	`abc\n\n`
`abc\n`	`abc\n`

In previous products, writing a single new-line character to a new file created an empty file under MVS. z/OS Language Environment treats a single new-line characters written to a new file as a special case, because it is the last new-line character of the file. The library writes a single blank to the file. When you read this file, you see two new-line characters instead of one. You also see two new-line characters on a read if you have written two new-line characters 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 7. 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

Table 7. Appending to ASA Files (continued)

abc Written to File, fclose() then append xyz	**What You Read from File after fclose(), fopen()**	
	Previous release	**New 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\rxyz\n	same as previous release

- The behavior of DBCS strings has changed.
  1. I/O now checks the value of `MB_CUR_MAX` to determine whether to interpret DBCS characters within a file.
  2. 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.
  3. 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.
  4. 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.
  5. 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`.

Repositioning within Files

- 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 new _EDC_COMPAT environment variable so thatsource code need not change to compensate for the new behavior. _EDC_COMPAT is described in *z/OS C/C++ Programming Guide*.
- 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. _EDC_COMPAT is described in *z/OS C/C++ Programming Guide*.
- 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. _EDC_COMPAT is described in *z/OS C/C++ Programming Guide* .
- If you have used ungetc() to move the file pointer to a position before the beginning of the file, calls to ftell() and fgetpos() now fail. Previously, ftell() returned the value 0 for such calls, but set errno to a non-zero value. Previously, fgetpos() did not account for ungetc() calls. See *z/OS C/C++ Programming Guide* for information on how to change fgetpos() behavior by using _EDC_COMPAT.

  For example, suppose that you are at relative position 1 in the file and ungetc() is performed twice. ftell() and fgetpos() will now report the relative position -1, which is before the start of the file, causing both ftell() and fgetpos() to fail.
- After you have called ftell(), calls to setbuf() or setvbuf() may fail. Applications should never call I/O functions between calls to fopen() or freopen() and calls to the functions that control buffering.

Closing and Reopening ASA Files

The behavior of ASA files when you close and reopen them is now consistent:

Table 8. Closing and Reopening ASA Files

	Physical record after close			
Written to file	**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)	

Table 8. Closing and Reopening ASA Files (continued)

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

fldata() Return Values

There are minor changes to the values that the `fldata()` library function returns. It may now return more specific information in some fields. For more information on `fldata()`, see the “Input and Output” section in *z/OS C/C++ Programming Guide*.

Error Handling

The general return code for errors is now `EOF`. In previous products, some I/O functions returned 1 as an error code to indicate failure. This caused some confusion, as 1 is a possible `errno` value as well as a return code. `EOF` is not a valid `errno` value.

Programs that rely on specific values of `errno` may not run as expected, because certain `errno` values have changed. Starting with OS/390 Language Environment V1R5, error messages have the format EDC5xxx. You can find the error message information for a particular `errno` value by applying the `errno` value to EDC5xxx (for example, 021 becomes EDC5021), and looking up the EDC5xxx message in *z/OS Language Environment Debugging Guide* manual.

Miscellaneous

- The inheritance model for standard streams now 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
`L_tmpnam`	47	1024
`FILENAME_MAX`	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.

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.

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.
- 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.
- 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.

- 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 4. From OS/390 C/C++ to z/OS V1R2 C/C++

This part discusses the implications of migrating applications that were created with one of the following compilers and one of the following libraries to the z/OS V1R2 C/C++ product.

Compilers:

- IBM OS/390 C/C++ V1R2 compiler, 5645-001
- IBM OS/390 C/C++ V1R3 compiler, 5645-001
- IBM OS/390 C/C++ V2R4 compiler, 5647-A01
- IBM OS/390 C/C++ V2R6 compiler, 5647-A01
- IBM OS/390 C/C++ V2R9 compiler, 5647-A01
- IBM OS/390 C/C++ V2R10 compiler, 5647-A01
- IBM z/OS C/C++ V1R1 compiler, 5694-A01

Libraries:

- OS/390 V1R2 Language Environment, 5645-001
- OS/390 V1R3 Language Environment, 5645-001
- OS/390 V2R4 Language Environment, 5647-A01
- OS/390 V2R5 Language Environment, 5647-A01
- OS/390 V2R6 Language Environment, 5647-A01
- OS/390 V2R7 Language Environment, 5647-A01
- OS/390 V2R8 Language Environment, 5647-A01
- OS/390 V2R9 Language Environment, 5647-A01
- OS/390 V2R10 Language Environment, 5647-A01
- IBM z/OS V1R1 Language Environment, 5694-A01

Notes:

1. The z/OS V1R1 compiler and library are equivalent to the OS/390 V2R10 compiler and library.
2. The OS/390 V1R1 compiler and library were equivalent to the final MVS/ESA compiler and library, and are described in Part 3 of this book.

Chapter 12. Compiler Changes Between OS/390 C/C++ and z/OS V1R2 C/C++

This chapter describes the compiler changes that you may encounter if you are migrating from a previous release of OS/390 C/C++ or z/OS V1R1 C/C++ to z/OS V1R2 C/C++. Also refer to “Chapter 15. OS/390 V2R10 to z/OS V1R2 C/C++ Compiler Changes” on page 93 for details on ISO C++ 1998 Standard Language support.

Compiler

Memory Consideration

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.

Removal of Model Tool Support

As of OS/390 V2R10, the Model Tool is no longer available.

Supporting the ISO standard

As of z/OS V1R2, the C/C++ compiler supports the C++ Standard. See “Chapter 15. OS/390 V2R10 to z/OS V1R2 C/C++ Compiler Changes” on page 93 for details. Note that the OS/390 V2R10 compiler and z/OS V1R2 compiler support different language levels; hence, they are both shipped as part of z/OS V1R2.

Pragma reachable and leaves

These pragmas 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 now more aggressive. Functions that exhibit the leave and reachable properties must be identified by these pragmas.

The C run-time library functions `setjmp` and `longjmp` (and the related `sigsetjmp`, `siglongjmp`, and so on) are such functions.

If your version of `setjmp.h` does not include these pragmas, you should add them to your program code as follows:

```
#pragma leaves (longjmp, _longjmp, siglongjmp)
#pragma reachable (setjmp, _setjmp, sigsetjmp)
```

Alternatively, if the functions refer to the C run-time library provided by the system (or another library that strictly conforms to the C standard), you can turn on the `LIBANSI` option.

For more information on using `#pragma reachable` and `#pragma leaves` directives, refer to *C/C++ Language Reference*.

Reentrant 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.

From OS/390 C/C++ to z/OS V1R2 C/C++

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. 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 with the z/OS V1R2 compiler).

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

Compiler Options

Compiler Options with Default Setting Changes

ARCHITECTURE Compiler Option: As of OS/390 V2R10, the default value of the ARCHITECTURE compiler option is 2, unless TARGET(OSV2R9 and below) is specified and then the default is 0. In previous releases, it was 0.

CHECKOUT(CAST) Compiler Suboption: This suboption instructs the C compiler to check the source code for pointer casting that might affect optimization, that is, those castings that violate the ansi-aliasing rule. (Refer to *z/OS C/C++ User's Guide* for more details about ANSIALIAS option.) Prior to z/OS V1R2, the compiler issues a WARNING message whenever this condition is detected. As of z/OS V1R2, this message is an INFORMATIONAL. If you wish to be alerted by the compiler that this message has been issued, you can use the HALTONMSG compiler option. (The message number is CCN3374.) The HALTONMSG option causes the compiler to stop after source code analysis, skip the code generation, and return with a return code of 12.

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

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

INLINE Compiler Option: 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.

OPTIMIZE Compiler Option: In the OS/390 C/C++ V1R2, V1R3, and V2R4 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:

- `OPT(0)` maps to `NOOPT`

- `OPT`, `OPT(1)` and `OPT(2)` map to `OPT(2)`

ROSTRING Compiler Option: As of z/OS V1R2, the `ROSTRING` option default for C and C++ is changed from `NOROSTRING` to `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.

TARGET Compiler Option: As of z/OS V1R2 the TARGET option only supports the following release sub-options: OSV2R6, OSV2R7, OSV2R8, OSV2R9, OSV2R10, zOSV1R1, and zOSV1R2. Earlier versions of the compiler also supported the following release sub-options: OSV1R2, OSV1R3, OSV2R4, and OSV2R5. This allows you to compile an application using the current compiler, and then link and run the application on a lower level system.

The new z/OS V1R2 C++ compiler supports the TARGET option, but may not generate the same code from previous releases using similar TARGET option settings, since the compilers support different levels of Standard C++. In view of this, the z/OS V1R1 C/C++ compiler (equivalent to OS/390 V2R10 C/C++ compiler) is also shipped as part of z/OS V1R2. The OS/390 V2R10 C/C++ compiler must be installed and explicitly invoked during compilation, as needed. Refer to “Setup Considerations” on page 94 on invoking the compiler.

New Compiler Option That May Affect Source Code

ENUM Compiler Option: z/OS V1R2 introduces the `ENUM` option as a means for controlling the size of enumeration types. The default setting, ENUM(SMALL), provides the same behavior in previous releases of the compiler. If you want to use the `ENUM` option, it is recommended that the same setting be used for the whole application; otherwise, you may 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).

Compiler Options That are No Longer Supported

As of z/OS V1R2, 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, the following SOM-related compiler options are no longer supported:

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

As of OS/390 V2R4, the IDL compiler option is no longer available. If you continue to require IDL for your applications, new IDL or IDL modifications must be coded by hand. You can then use the IDL compiler to generate your C/C++ source code.

Compiler Messages and Return Codes

There are differences in messages and return codes between different versions of the compiler. Message contents have changed, and return codes for some messages have changed (some errors have become warnings, and in very rare situations, some warnings have become errors). You must update accordingly any application that is affected by message contents or return codes. **Do not build dependencies on message content, message numbers, or return codes**. See *z/OS C/C++ Messages* for a list of compiler messages.

Changes in Data Set Names

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

Compiler Listings

As of OS/390 C/C++ V2 R6, `OPT(1)` maps 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 listings for the current release, refer to the *z/OS C/C++ User's Guide*.

Changes That Affect User JCL

Specifying Class Library Header Files at Compile Time

In OS/390 V2R10 and earlier compilers, using the following JCL on the compile step would work, although not recommended:

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

As of z/OS V1R2, the record size for the SCLBH data sets have been increased from LRECL=80 to LRECL=120. Due to this change, the SYSLIB shown above will no longer work, and must be removed from your JCL. The replacement for this is the `SEARCH` compiler option, as in the following example:

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

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

Interprocedural Analysis

IPA Object Module Binary Compatibility

Release-to-release binary compatibility is maintained by the z/OS C/C++ IPA Compile and IPA Link as follows:

- An object file produced by an IPA Compile which contains IPA Object or combined IPA and conventional object information can be used as input to the IPA Link of the same or later Version/Release of the compiler.
- An object file produced by an IPA Compile which contains IPA Object or combined IPA and conventional object information cannot be used as input by the IPA Link of an earlier Version/Release of the compiler. If this is attempted, an error diagnostic message will be issued by the IPA Link.
- Note that if the IPA object is recompiled by a later IPA Compile, additional optimizations may be performed and the resulting application program may perform better.

An exception to this is the IPA object files produced by the OS/390 Release 2 C IPA Compile. These must by recompiled from the program source using an OS/390 V1R3 or later compiler before attempting to process them with the z/OS V1R2 C/C++ IPA Link.

IPA Link Step Defaults

As of OS/390 V1R3, 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.
- The default inlining threshold is now 1000 ACUs. With OS/390 C/C++ V1R2, the threshold was 100 ACUs.
- The default expansion threshold is now 8000 ACUs. With OS/390 C/C++ V1R2, the threshold was 1000 ACUs.

As of OS/390 V2R6, the default optimization level for the IPA Link step is `OPT(2)`.

Data Types

Floating Point Type to Integer Conversion

Consider the following piece of code where a floating point type is converted to a signed integer type:

```
double x;
int i;
/* ... */

i = x; /* overflow if x is too large */
/* value of i undefined */
```

When the 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, it should be checked explicitly. For example:

```
double x;
int i;
if (x < (double) INT_MAX)
    i = x;
else {
    /* overflow */
}
```

Long Long Data Type

As of OS/390 V2R9, the C/C++ compiler and Language Environment support long long data types. The _LONG_LONG macro is predefined for all language levels other than ANSI.

As of z/OS V1R2, long long 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)`. If you have defined your own _LONG_LONG macro in previous compiler releases, you must remove this user-defined macro before compiling a program in z/OS V1R2. When `LANGLVL(LONGLONG)` is turned on the `_LONG_LONG` macro is defined by the compiler.

Chapter 13. Language Environment Changes Between OS/390 C/C++ and z/OS V1R2 C/C++

This chapter describes the Language Environment elements that may impact your migration from a previous release of OS/390 C/C++ or z/OS V1R1 C/C++ to z/OS V1R2 C/C++.

Name Conflicts with Run-Time Library Functions

When taking code previously compiled and link-edited on a system earlier than OS/390 V2R4, and moving to a system at OS/390 V2R4 or later, you might have a problem with name conflicts if both the following are true:

1. You created functions with the same name as library functions.
2. When linking your application you included the IBM supplied Language Environment link library before the files that contain your function definitions.

Previous releases of the OS/390 C/C++ run-time headers used the #pragma map directive to convert many function names into identifiers prefixed with "@@". For example, if you included `fcntl.h` in your source, a reference to `open()` in your source code resulted in an external name `@@OPEN` in the object code. As of OS/390 V2R4 many pragma maps have been eliminated. If you created functions with the same name as library functions, you must ensure that the file containing your version of the function precedes the IBM supplied Language Environment link library in the search order when linking your application. If you have object modules containing identifiers like `OPEN` that you want resolved to your version of `open()`, you may need to alter your JCL to ensure that your version precedes the IBM supplied Language Environment link library in the search order.

Also, if you have multiple, interdependent modules that rely on the name mapping present in prior releases, you cannot recompile one without recompiling the others. For example, module A includes `fcntl.h` and calls `open()` resulting in a reference to `@@OPEN` in the object code. Module B implements your version of `open()` and also includes `fcntl.h`, so that the external name of the called function is mapped to `@@OPEN`. You must recompile both modules.

Table 9 lists the functions that had pragma maps deleted in OS/390 V2R4.

Table 9. Functions That Had Pragma Maps Deleted

____loc1()	__atoe()	__atoe_l()	__cnvblk()	__dlght()
__etoa()	__etoa_l()	__gderr()	__getipc()	__ipdbcs()
__ipdspx()	__iphost()	__ipmsgc()	__ipnode()	__iptcpn()
__opargf()	__operrf()	__opindf()	__opoptf()	__sigerr()
__sigign()	__sigpro()	__tzone()	__wsinit()	_longjmp()
_setjmp()	_tolower()	_toupper()	accept()	access()
alarm()	a64l()	basename()	bcmp()	bcopy()
bind()	brk()	bzero()	catclose()	catgets()
catopen()	cclass()	chaudit()	chdir()	chmod()
chown()	chroot()	clearenv()	clearenv()	close()
closedir()	closelog()	clrmemf()	confstr()	connect()
creat()	crypt()	ctdli()	ctdli()	ctermid()
ctermid()	cuserid()	cuserid()	dirname()	drand48()
dup()	dup2()	dynalloc()	dynfree()	ecvt()
encrypt()	endgrent()	endpwent()	erand48()	execl()
execle()	execlp()	execv()	execve()	execvp()

Table 9. Functions That Had Pragma Maps Deleted (continued)

fattach()	fchaudit()	fchdir()	fchmod()	fcntl()
fcvt()	fdelrec()	fdetach()	fetch()	fetchep()
ffs()	fileno()	fldata()	flocate()	fmtmsg()
fnmatch()	fork()	fstat()	fstatvfs()	ftime()
ftok()	ftw()	fupdate()	gcsp()	gcvt()
getcwd()	getdate()	getegid()	geteuid()	getgid()
getgrent()	getgrgid()	getgrnam()	getmsg()	getopt()
getopt()	getpass()	getpgid()	getpgrp()	getpid()
getpmsg()	getppid()	getpwent()	getpwnam()	getpwuid()
getsid()	getsyntx()	getuid()	getutxid()	getw()
getwd()	glob()	globfree()	grantpt()	hcreate()
hdestroy()	hsearch()	iconv()	index()	insque()
ioctl()	ioctl()	isatty()	isnan()	jrand48()
kill()	killpg()	lchown()	lcong48()	lfind()
link()	listen()	lockf()	lrand48()	lsearch()
lseek()	lstat()	l64a()	maxcoll()	maxdesc()
memccpy()	mkdir()	mkfifo()	mkstemp()	mktemp()
mmap()	mount()	mprotect()	mrand48()	msgctl()
msgget()	msgrcv()	msgsnd()	msgxrcv()	msync()
munmap()	nftw()	nice()	nlist()	nrand48()
open()	opendir()	openlog()	pathconf()	pause()
pclose()	pipe()	poll()	popen()	ptsname()
putenv()	putmsg()	putpmsg()	putw()	random()
re_comp()	re_exec()	read()	readdir()	readv()
realpath()	recv()	recvfrom()	regcmp()	regcomp()
regerror()	regex()	regexec()	regfree()	release()
remque()	rexec()	rindex()	rmdir()	sbrk()
scalb()	seed48()	seekdir()	semctl()	semget()
semop()	send()	sendto()	setegid()	setenv()
setenv()	seteuid()	setgid()	setgrent()	setkey()
setpeer()	setpgid()	setpgrp()	setpwent()	setregid()
setreuid()	setsid()	setstate()	setuid()	shmat()
shmctl()	shmdt()	shmget()	shutdown()	sighold()
sigpause()	sigrelse()	sigset()	sigstack()	sigwait()
sleep()	socket()	spawn()	spawnp()	srandom()
srand48()	stat()	statvfs()	strdup()	strfmon()
strptime()	svc99()	swab()	sync()	sysconf()
syslog()	t_accept()	t_alloc()	t_bind()	t_close()
t_error()	t_free()	t_listen()	t_look()	t_open()
t_rcv()	t_rcvdis()	t_rcvrel()	t_snd()	t_snddis()
t_sndrel()	t_sync()	t_unbind()	tcdrain()	tcflow()
tcflush()	tcgetsid()	tdelete()	telldir()	tempnam()
tfind()	times()	tinit()	truncate()	tsearch()
tsetsubt()	tsyncro()	tterm()	ttyname()	ttyslot()
twalk()	tzset()	ualarm()	ulimit()	umask()
umount()	uname()	unlink()	unlockpt()	usleep()
utime()	utimes()	utimes()	valloc()	vfork()
w_ioctl()	w_statfs()	wait()	waitid()	waitpid()
wait3()	wordexp()	wordfree()	write()	writev()

Time Functions

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 C/C++ Programming Guide* for more information about both environment variables and customizing a locale.

Direct UCS-2 and UTF-8 Converters

OS/390 V2R9 added new UCS-2 and UTF-8 converters. These are direct conversions that no longer use the tables built by the uconvdef utility processing of UCMAPS. If you have modified UCMAPS, UCS-2 and UTF-8 converters will no longer use those modified UCMAPS. If you still need to use the modifications that you made to UCMAPS, you will now need to set the _ICONV_UCS environment variable to "0". Refer to *z/OS C/C++ Programming Guide* for more information about the _ICONV_UCS environment variable.

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 CEEDOPT (CEECOPT for CICS). Refer to *z/OS Language Environment Customization* for details on changing CEEDOPT and CEECOPT.

THREADSTACK Run-Time Option

As of OS/390 V2R10 Language Environment the new THREADSTACK run-time option replaces the NONIPTSTACK and NONONIPTSTACK options. The old options will still be accepted, but an information message will be issued, telling the user to switch to the new THREADSTACK option. The old options do not have support for specifying the initial and increment sizes of the new XPLINK downward growing stack. Refer to *z/OS Language Environment Customization* for more information on the THREADSTACK run-time option.

Chapter 14. Class Library Changes Between OS/390 C/C++ and z/OS V1R2 C/C++

If you are using class libraries, this chapter describes the changes that you may have to make if you are migrating from a previous release of OS/390 C/C++ or z/OS V1R1 C/C++, to z/OS V1R2 C/C++. Also refer to “Chapter 15. OS/390 V2R10 to z/OS V1R2 C/C++ Compiler Changes” on page 93 for details on ISO C++ 1998 Standard language support.

z/OS V1R2 IBM Open Class Library

The z/OS V1R2 IBM Open Class Library and OS/390 V2R10 IBM Open Class Library are both shipped in z/OS V1R2. DLLs and static libraries for each version are shipped in different data sets. Side-decks as well as header files reside in common data sets.

The table below shows the different ways by which you can migrate your C++ application that uses class libraries.

Table 10. Typical Migration in z/OS Involving Class Libraries

Application to be Ported to z/OS V1R2		**Setup in z/OS V1R2**	
Current Location	**Class Library In Use**	**Compiler to Use**	**Class Library to Use**
Another platform such as VisualAge® C++ for AIX® Version 5.0	C++ Standard Library	z/OS V1R2	C++ Standard Library
OS/390 V2R10	Third party STLPORT		
Another platform such as VisualAge C++ for AIX Version 5.0	IBM Open Class Library Version 5 for AIX	z/OS V1R2	z/OS V1R2 IBM Open Class Library **Note:** This class library is consistent with that shipped in VisualAge C++ for AIX V5.0. It is intended to ease porting from AIX, but is not intended for use in new development. Support will be withdrawn in a future release.
OS/390 V2R10	OS/390 V2R10 IBM Open Class Library	z/OS V1R2	z/OS V1R2 IBM Open Class Library
OS/390 V2R10	OS/390 V2R10 IBM Open Class Library	z/OS V1R2 with `TARGET(OSV2R10)`	OS/390 V2R10 IBM Open Class Library
Another platform such as VisualAge C++ for AIX Version 3.6.6	IBM Open Class Library Version 3 for AIX	• z/OS V1R2 with `TARGET(OSV2R10)`, or • OS/390 V2R10	OS/390 V2R10 IBM Open Class Library

The following table summarizes the components of both versions of the IBM Open Class Libraries.

Table 11. Components of Class Libraries

Class Library Component	C++ Standard Library	z/OS V1R2 IBM Open Class Library	OS/390 V2R10 IBM Open Class Library
Header Files	CEE.SCEEH*	• CBC.SCLBH.H • CBC.SCLBH.HPP • CBC.SCLBH.INL • CBC.SCLBH.C	• CBC.SCLBH.H • CBC.SCLBH.HPP • CBC.SCLBH.INL • CBC.SCLBH.C
Side-Decks	CEE.SCEELIB • C128 **Note:** This is shipped with Language Environment in z/OS V1R2, and provides new versions of the USL IOSTREAM and COMPLEX libraries. For portable code, use the new versions.	CBC.SCLBSID • IOC • COLL	CBC.SCLBSID • ASCCOLL • APPSUPP • COLLECT • COMPLEX • IOSTREAM
Static Libraries		CBC.SCLBCPP2 • XPLINK IOC static library • non-XPLINK IOC static library • XPLINK COLL static library • non-XPLINK COLL static library • XPLINK USL IOSTREAM static library • non-XPLINK USL IOSTREAM static library • XPLINK USL COMPLEX static library • non-XPLINK USL COMPLEX static library	CBC.SCLBCPP • XPLINK APPSUPP static library • non-XPLINK APPSUPP static library • XPLINK COLLECT static library • non-XPLINK COLLECT static library • XPLINK USL IOSTREAM static library • non-XPLINK USL IOSTREAM static library • XPLINK USL COMPLEX static library • non-XPLINK USL COMPLEX static library
DLL data sets	CEE.SCEERUN	CBC.SCLBDLL2 (XPLINK only) • IOC • COLL	CBC.SCLBDLL (non-XPLINK only) • ASCCOLL • APPSUPP • COLLECT • COMPLEX • IOSTREAM

One set of headers are shipped in CBC.SCLBH.* datasets for both z/OS V1R2 IBM Open Class Library and OS/390 V2R10 IBM Open Class Library. If a header file

with the same name exists in both versions of the IBM Open Class, then code that is applicable for both versions is present in the file, but separated by an appropriate TARGET release guard. When using this type of header with the z/OS V1R2 compiler to build applications targetted to previous OS/390 releases, the OS/390 V2R10 header content can be invoked by using TARGET compiler option, for example, TARGET(OSV2R10).

The side-decks of z/OS V1R2 IBM Open Class and OS/390 V2R10 IBM Open Class reside in a common dataset, CBC.SCLBSID. Use side-deck members from one version only, for any given environment; otherwise, unexpected behavior may occur as a result of symbol conflicts in these side-decks.

The DLLs of z/OS V1R2 IBM Open Class reside in the CBC.SCLBDLL2 data set. The DLLs of OS/390 V2R10 IBM Open Class reside in the CBC.SCLBDLL dataset. If you are using the z/OS V1R2 IBM Open Class Library (which uses the USL I/O Stream Library), you must specify both CBC.SCLBDLL2 and CBC.SCLBDLL data sets.

The DLLs for the z/OS V1R2 IBM Open Class Library are compiled with XPLINK. If you use these DLLs with non-XPLINK applications, you may notice further degradation of performance (depending on the frequency of use of DLL functions) due to the cost of switching to the XPLINK environment when crossing the boundary between your application code and the class library code in the DLL. To avoid this problem, use the static library instead of the DLL. This static library has both the XPLINK and NOXPLINK versions of the objects.

The static libraries of z/OS V1R2 IBM Open Class reside in CBC.SCLBCPP2 data set. The static libraries of OS/390 V2R10 IBM Open Class reside in CBC.SCLBCPP data set. Note that the USL IOSTREAM and COMPLEX static libraries have copies in both CBC.SCLBCPP2 and CBC.SCLBCPP data sets. Applications built for z/OS V1R2 require CBC.SCLBCPP2 only in the environment settings. Applications built for OS/390 require CBC.SCLBCPP only in the environment settings.

Compile-Time Requirements for z/OS V1R2 IBM Open Class Library

- To use the z/OS V1R2 IBM Open Class Library, use the z/OS V1R2 C++ compiler and do not specify the TARGET compiler option. Use the definition sidedecks IOC and COLL, which correspond to the DLLs in the CBC.SCLBDLL2 data set. If you want to use the static library, it resides in the CBC.SCLBCPP2 data set.
- When using the z/OS V1R2 IBM Open Class Library header files:
  - Use the LANGLVL(EXTENDED) compiler option and do not turn off any suboptions related to this option.
  - Ensure that the following macros are defined:
    - _AE_BIMODAL
    - _ALL_SOURCE
    - _MSE_PROTOS
    - _OPEN_SOURCE=3

      Note: Setting _OPEN_SOURCE macro to another value may result in unexpected behavior.
    - _SHR_ENVIRON

See the section about compiling IBM Open Class applications in *IBM Open Class Library User's Guide* for more details.

Runtime Requirements for z/OS V1R2 IBM Open Class Library

The z/OS V1R2 IBM Open Class Library requires the `POSIX(ON)` run-time option to be specified. This option is required for the POSIX threading functions, and the system and signal handling functions. An application using file systems classes from z/OS V1R2 IBM Open Class must run on a system where the z/OS UNIX kernel is available and active, as the implementations of file systems classes are limited to UNIX Systems Services.

The z/OS V1R2 IBM Open Class DLLs are built with XPLINK. All the DLLs from the previous versions/releases are built with NOXPLINK, including the OS/390 V2R10 DLLs which are available in z/OS V1R2. If you migrate to the upgraded IBM Open Class DLLs and your application is not built XPLINK, then you must specify the XPLINK(ON) run-time option, which may cause a performance degradation to the application. If you are not using XPLINK, then you should continue to use the OS/390 V2R10 IBM Open Class DLLs by either using the OS/390 V2R10 C++ compiler or the z/OS V1R2 C++ compiler with the `TARGET(OSV2R10)` compiler option specified. Alternatively, you can use the static version of the upgraded IBM Open Class Library by linking in the static library CBC.SCLBCPP2 which contains both XPLINK and NOXPLINK objects.

OS/390 V2R10 IBM Open Class Library

To use the OS/390 V2R10 IBM Open Class Library, use the z/OS V1R2 C++ compiler and specify the `TARGET(OSV2R10)` compiler option or use the OS/390 V2R10 C++ compiler. Use the definition side-decks ASCCOLL, APPSUPP, and/or COLLECT, which correspond to the DLLs in the CBC.SCLBDLL data set. If you want to use the static library, it resides in the CBC.SCLBCPP data set.

The OS/390 V2R10 IBM Open Class DLLs are compiled with `NOXPLINK`. If you use these DLLs with `XPLINK` applications, the performance gain you realize in your application code by using `XPLINK` may be offset partially or completely (depending on the frequency of use of DLL functions) by the cost of switching to the non-`XPLINK` environment when crossing the boundary between your application code and the class library code in the DLL. If you use these DLLs with `XPLINK` applications, you may notice reduced performance. You can avoid this by using the static library instead of the DLL. This static library has both the `XPLINK` and `NOXPLINK` versions of the objects, and resides in the CBC.SCLBCPP data set.

Migrating from USL I/O Stream Library to C++ Standard I/O Stream Library

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

The following flags have been added:

- flags for controlling formatting: `boolalpha, adjustfield, basefield, floatfield`

The following flags have been removed:

- flags for controlling formatting: `stdio`
- flags for controlling the open mode: `nocreate, noreplace, bin`
- flags for controlling the io state: `hardfail`

There may be other small differences.

Mixing the C++ Standard I/O Stream Library, USL I/O Stream Library, and C I/O

While it is possible to mix the usage of the C++ Standard I/O Stream Library, the USL I/O Stream Library, and C I/O, it is not recommended. The USL I/O Stream Library uses a separate buffer so you would need to flush the buffer after each call to `cout` by either setting `unitbuf` or `sync_with_stdio()`. You should avoid switching between the I/O Stream Library formatted extraction functions and C stdio library functions whenever possible, and you should also avoid switching between versions of the I/O Stream Libraries. For more information see the part that discusses input and output in *z/OS C/C++ Programming Guide*.

The z/OS V1R2 IBM Open Class Library includes a new header, `istream.hpp`. This header allows you to choose the I/O Stream library you want to use through a macro, `__IOC_ANSI_STREAM`. For example, a truly portable IBM Open Class application is written this way:

```
#include <istring.hpp>
#ifdef __IOC_ANSI_STREAM
    #include <iostream>
    #define _std_ std::
#else
    #include <iostream.h>
    #define _std_
#endif

int main(void)
{
  IString is("I can hear music");
  _std_ cout << is << _std_ endl;
  return 0;
}
```

In your makefile, define `__IOC_ANSI_STREAM` if you want to use the C++ Standard I/O Stream Library. By default, the IBM Open Class Library uses the USL I/O Stream Library to preserve existing behavior.

Class Library Execution and Object Module Incompatibilities

There are execution and object module incompatibilities between the class libraries provided in z/OS V1R2 and earlier releases. Unless you are using the OS/390 V2R10 IBM Open Class Library, which also ships in z/OS V1R2, you must recompile and relink applications that are dynamically bound to those class libraries.

Also refer to “Appendix. Class Library Migration Considerations” on page 109 for some background information about class libraries and compatibility considerations.

Collection Class Library Source Code Incompatibilities

There are source code incompatibilities between the native Collection Class Libraries available with z/OS V1R2 and previous versions/releases. You must change your source code if you are migrating to the z/OS V1R2 IBM Open Class Library from an earlier version/release of the IBM Open Class Library.

Also, the structure of the Collection Classes is changed in z/OS V1R2. All classes, including the concrete classes, are now related in an abstract hierarchy. The abstract hierarchy makes use of virtual inheritance. When you subclass from a Collection Class and implement your own copy constructor, you must initialize the

virtual base class IACollection<Element> in your derived classes. Therefore, if you subclassed from a concrete Collection Class that was shipped with OS/390 V2R10 C++, and are migrating to the Collection Classes that are shipped with z/OS V1R2 C++, you will have to change the implementation of your copy constructor by adding the virtual base class initialization. Refer to the chapter about "Collection Class Changes", in the *IBM Open Class Library User's Guide* for more information.

Removal of SOM™ Support

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

Removal of Database Access Class Library Utility

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

Part 5. ISO C++ 1998 Standard Migration Issues

This part discusses the implications of migrating applications that were created with z/OS V1R1 C/C++ compiler or earlier, to the z/OS V1R2 C/C++ which is compliant with ISO C++ 1998 Standard.

Note: The z/OS V1R1 compiler and library are equivalent to the OS/390 V2R10 compiler and library.

Chapter 15. OS/390 V2R10 to z/OS V1R2 C/C++ Compiler Changes

The OS/390 V2R10 C/C++ (also shipped in z/OS V1R1) is shipping as part of the z/OS V1R2 operating system along with the new z/OS V1R2 C/C++ compiler. The OS/390 V2R10 compiler continues to ship in the CBC.SCBC* data sets. The new z/OS V1R2 compiler ships in the CBC.SCCN* data sets. This is being done as a migration aid for incompatibilities introduced during the implementation of the latest ISO C++ Standard. This chapter discusses the incompatibilities due to changes between OS/390 V2R10 and z/OS V1R2 compilers.

To serve the unique needs of each C++ customer, the available migration options are summarized in “Choosing an Approach Based on Migration Objectives”, “Setup Considerations” on page 94, and “Options for Compatibility Between C++ Compilers” on page 95. Both compilers are installed by default; you can choose whether to license and enable one or the other or both. If you choose to install both, each compiler must be installed separately and explicitly invoked during compilation.

You may encounter situations in which code that compiles without errors in earlier C++ compilers, may produce warnings or error messages in the z/OS V1R2 C++ compiler. This could be due to changes in the language or due to differences in the compiler behavior. If you are migrating from the OS/390 V2R10 C++ compiler and earlier compilers, you should be aware of potential syntax errors with the z/OS V1R2 C++ compiler.

Language elements that may affect your code are provided in “Changes in Language Features to Support ISO C++ 1998 Standard” on page 96 and “New Language Features to Support ISO C++ 1998 Standard” on page 98. The “Examples of Errors Due to Changes in Compiler Behavior” on page 100 may also be applicable to you.

Choosing an Approach Based on Migration Objectives

Table 12 shows the different migration scenarios and the recommended approach for each. Choose the best fit for your installation.

Table 12. Migration objectives and approaches

Where you are	**Where you want to be**	**How to get there**	
		z/OS Implementation	**Choice of compiler**
Code is Standard C++ compliant (ported or new)	Remain compliant with Standard C++	Use LANGLVL(ANSI)	Install z/OS V1R2 C/C++ (prefix CCN)

Table 12. Migration objectives and approaches (continued)

Where you are	Where you want to be	How to get there	
		z/OS Implementation	**Choice of compiler**
Code compiles with OS/390 V2R10	• Exploit Standard C++ language features • Willing to modify codebases to exploit Standard C++	Use the following to aid the migration process: • LANGLVL(COMPAT92) • LANGLVL() suboptions to control individual language features	Install z/OS V1R2 C/C++ (prefix CCN)
Code is compiling on OS/390 V2R10 C++	• No need for Standard C++ language features • Do not want to modify codebases	Try the following: • Use LANGLVL(COMPAT92) to tolerate language incompatibilities • Use OS/390 V2R10 C++[1]	Install either or both: • z/OS V1R2 C/C++ (prefix CCN) • OS/390 V2R10 C/C++ (prefix CBC)
Note: [1]The OS/390 V2R10 C/C++ compiler is being shipped as part of z/OS V1R2 to aid in migration. In a future release, the OS/390 V2R10 compiler will not be shipped as part of the operating system.			

Setup Considerations

Table 13 lists the different components for z/OS V1R2 and OS/390 V2R10. These must be setup based on your migration path from “Choosing an Approach Based on Migration Objectives” on page 93.

Table 13. Setup considerations for z/OS V1R2 and OS/390 V2R10 compilers

Component	Setup Considerations	Value for z/OS V1R2 C/C++ (Default)	Value for OS/390 V2R10 C/C++
z/OS UNIX®	_C89_CVERSION	″0x41020000″	″0x220A0000″
	_CC_CVERSION	″0x41020000″	″0x220A0000″
	_CXX_CVERSION	″0x41020000″	″0x220A0000″
	_CXX_CLASSVERSION	″0x41020000″	″0x220A0000″
	_C89_CNAME	″CCNDRVR″	″CBCDRVR″
	_CC_CNAME	″CCNDRVR″	″CBCDRVR″
	_CXX_CNAME	″CCNDRVR″	″CBCDRVR″
	STEPLIB	include SCCNCMP	include SCBCCMP
JCL	PROCLIBS: Switch compilers by overriding PROCLIBs.	PROCS in SCCNPRC : The PROC names are the same as in SCBCPRC so both sets of PROCS cannot reside in SYS1.PROCLIB.	PROCS in SCBCPRC

Table 13. Setup considerations for z/OS V1R2 and OS/390 V2R10 compilers (continued)

Component	Setup Considerations	Value for z/OS V1R2 C/C++ (Default)	Value for OS/390 V2R10 C/C++
TSO	REXX EXECs : Switch compilers by overriding SYSPROC libs and STEPLIBs.	SCCNUTL: The utility names are the same as in SCBCUTL so both sets of REXX EXECs cannot be concatenated together.	SCBCUTL

Options for Compatibility Between C++ Compilers

You can control individual language features in the z/OS V1R2 C++ compiler by using the `LANGLVL` and `KEYWORD` suboptions listed in Table 14. (Please refer to the *z/OS C/C++ User's Guide* for details.) In order to conform to the C++ Standard, you may need to make a number of changes to your existing source code. These suboptions can help by breaking up the changes into smaller steps.

Three predefined option groups are provided for commonly-used settings. These groups are:

`LANGLVL(COMPAT92)`
Use this option group if your code compiles with OS/390 V2R10 C/C++ and you want to move to z/OS V1R2 C/C++ with minimal changes. This group is the closest you can get to the old behavior of the previous compilers.

`LANGLVL(STRICT98)` or `LANGLVL(ANSI)`
These two groups are identical. Use them if you are compiling new or ported code that is Standard C++ compliant.

`LANGLVL(EXTENDED)`
This option group indicates all language constructs available with z/OS C/C++. This enables extensions to the C++ Standard.

The options and settings included in each group are listed in the table below. Except for TMPLPARSE, all settings have a value of either **On** (meaning the suboption is enabled) or **Off** (meaning the suboption is not enabled).

Table 14. Compatibility Options for z/OS V1R2 and OS/390 V2R10 Compilers

Options	Group names		
	compat92	strict98/ ansi	extended
KEYWORD(bool) | NOKEYWORD(bool)	Off	On	On
KEYWORD(explicit) | NOKEYWORD(explicit)	Off	On	On
KEYWORD(export) | NOKEYWORD(export)	Off	On	On
KEYWORD(false) | NOKEYWORD(false)	Off	On	On
KEYWORD(mutable) | NOKEYWORD(mutable)	Off	On	On
KEYWORD(namespace) | NOKEYWORD(namespace)	Off	On	On

Table 14. Compatibility Options for z/OS V1R2 and OS/390 V2R10 Compilers (continued)

Options	Group names		
	compat92	**strict98/ ansi**	**extended**
KEYWORD(true) | NOKEYWORD(true)	Off	On	On
KEYWORD(typename) | NOKEYWORD(typename)	Off	On	On
KEYWORD(using) | NOKEYWORD(using)	Off	On	On
LANGLVL(ANONSTRUCT | NOANONSTRUCT)	Off	Off	On
LANGLVL(ANONUNION | NOANONUNION)	On	Off	On
LANGLVL(ANSIFOR | NOANSIFOR)	Off	On	On
LANGLVL(ILLPTOMEM | NOILLPTOMEM)	On	Off	On
LANGLVL(IMPLICITINT | NOIMPLICITINT)	On	Off	On
LANGLVL(LIBEXT | NOLIBEXT)	On	Off	On
LANGLVL(LONGLONG | NOLONGLONG)	On	Off	On
LANGLVL(OFFSETNONPOD | OFFSETNONPOD)	On	Off	On
LANGLVL(OLDDIGRAPH | OLDDIGRAPH)	Off	On	Off
LANGLVL(OLDFRIEND | NOOLDFRIEND)	On	Off	On
LANGLVL(OLDMATH | NOOLDMATH)	On	Off	Off
LANGLVL(OLDTEMPACC | NOOLDTEMPACC)	On	Off	On
LANGLVL(OLDTMPLALIGN | NOOLDTMPLALIGN)	On	Off	Off
LANGLVL(OLDTMPLSPEC | NOOLDTMPLSPEC)	On	Off	On
LANGLVL(TRAILENUM | NOTRAILENUM)	On	Off	On
LANGLVL(TYPEDEFCLASS | TYPEDEFCLASS)	On	Off	On
LANGLVL(ZEROEXTARRAY | NOZEROEXTARRAY)	Off	Off	On
RTTI | NORTTI	Off	On	On
TMPLPARSE(NO | ERROR | WARN)	NO	WARN	WARN

Changes in Language Features to Support ISO C++ 1998 Standard

Refer to *C/C++ Language Reference* for more details.

for-loop Scoping

In Standard C++, the scope of variable in `for-loop` initializer declaration is to the end of the loop body. The scope of such variables in the OS/390 V2R10 compiler and earlier compilers, is to the end of the lexical block containing the `for-loop`. For example:

```
int i=0;

void f()
{
    for(int i=0; i<10; i++)
    {
       if(...) break;
    }
    if(i==10) { ... }      // 1
    ...
}
```

1

- As of z/OS V1R2, this means ::`i` (the `i` declared in file scope).
- In OS/390 V2R10 and earlier compilers, this means the local `i`. To maintain this context within z/OS V1R2, use `LANGLVL(NOANSIFOR)` option.

Implicit int Is No Longer Supported

The use of an implicit `int` in a declaration is no longer valid in Standard C++.

For example:

```
const i;   // previously meant const int i
main() { } // previously returned int
```

Hence, as of z/OS V1R2, the following code is no longer valid:

```
inline f() {
    return 0;
}
```

To comply with the standard, specify the type of every function and variable. Use the `LANGLVL(IMPLICITINT)` option to compile code containing implicit `int`s.

Changes to friend Declarations

As of the z/OS V1R2 C++ compiler, a class named as a `friend` is not visible until introduced into scope by another declaration:

```
class C {
    friend class D;
};
D* p;  // error, D not in scope
```

Friend class declarations must always be elaborated.

```
friend class C; // need class keyword
```

To allow friend declarations without elaborated class names, use `LANGLVL(OLDFRIEND)` option.

Exception Handling

In z/OS V1R2:

- A temporary copy is thrown rather than the actual object itself.

- The cv-qualification in the catch clause is not considered when the type caught is the same (possibly cv-qualified) type as that thrown or 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 OS/390 V2R10 and earlier compilers. There is no available option in z/OS V1R2 to enable the old behavior.

Operator new and delete

As of z/OS V1R2, operator new throws `std::bad_alloc` if memory allocation fails.

In the OS/390 V2R10 compiler and earlier compilers, operator new returned 0 to indicate allocation failure.

As of z/OS V1R2, an operator delete function that is a member of a class S must be accessible at the point at which the corresponding operator new class member function of S is invoked.

```
struct S {
    static void* operator new(size_t, void*);
  private:
    static operator delete(void*,size_t);
};

void f(void* p)
{
    new (p) S; // error, does not comply with standard
}
```

Note: The OS/390 V2R10 compiler and earlier compilers do not support overloaded delete.

New Language Features to Support ISO C++ 1998 Standard

New Keywords

As of z/OS V1R2, the following names are reserved as keywords and cannot be used for naming identifiers:

- bool
- explicit
- export
- false
- mutable
- namespace
- true
- typename
- using

If you are compiling old code that uses, for example, `typename` as an identifier, you can either remove your definition, or use the `NOKEYWORD(TYPENAME)` option.

Namespaces

Namespaces are not supported in the OS/390 V2R10 compiler and earlier compilers. Code ported to OS/390 V2R10 and earlier compilers from other platforms that support namespaces, may have implemented a workaround by defining it as a macro to nothing:

```
#define std
#define using
#define namespace
```

To compile such code in z/OS V1R2, you need to undefine the macro.

The bool Type

The `bool` type is not supported in the OS/390 V2R9 compiler and earlier compilers. Be aware that relational operators return bool in z/OS V1R2, while it returns int in previous compilers.

To disable this keyword in z/OS V1R2, use the `NOKEYWORD(BOOL)` option.

mutable Keyword

As of z/OS V1R2, the `mutable` keyword allows a class data member to be modified even though it is the data member of a `const` object.

This keyword is not supported in the OS/390 V2R10 compiler and earlier compilers. Code ported to OS/390 V2R10 and earlier compilers from other platforms, may have implemented a workaround by defining it as a macro to nothing:

```
#define mutable
```

To compile such code in z/OS V1R2, you need to undefine the macro.

To disable this keyword in z/OS V1R2, use the `NOKEYWORD(MUTABLE)` option.

wchar_t as Simple Type

The z/OS V1R2 compiler defines `wchar_t` as a simple type. The OS/390 V2R10 compiler and earlier compilers define it as a typedef.

explicit

The OS/390 V2R10 compiler and earlier compilers do not support the `explicit` keyword.

The purpose of this keyword is to make what would otherwise be a conversion constructor into a normal constructor. For example:

```
class C {
    explicit C(int);
};
C c(1);   // ok
C d = 1;  // error, no conversion constructor
```

To disable this keyword in z/OS V1R2, use the `NOKEYWORD(EXPLICIT)` option.

C++ cast

z/OS V1R2 has new cast operators: const_cast, dynamic_cast, reinterpret_cast and static_cast. These were not supported in the OS/390 V2R10 compiler and earlier compilers.

Changes to digraphs in the C++ Language

The ISO C++ standard now defines `and`, `bitor`, `or`, `xor`, `compl`, `bitand`, `and_eq`, `or_eq`, `xor_eq`, `not` and `not_eq` as alternate tokens for `&&`, `|`, `||`, `^`, `~`, `&`, `&=`,`|=`, `|=`, `!` and `!=`. As of z/OS V1R2, a program that uses any of these alternate tokens as variable, function, or type names, must be compiled with the `NODIGRAPH` option to suppress the parsing of these tokens as digraphs.

Examples of Errors Due to Changes in Compiler Behavior

The following are examples of code which compiles without errors in the OS/390 V2R10 compiler and earlier compilers, but which will produce errors or warnings in the z/OS V1R2 compiler. *z/OS C/C++ Messages* has more details on compiler messages.

Access-Checking Errors

```
class A {
    class B {
        void f(A::B);
        // A::B is private and cannot be accessed from B
        // void f(B); <--this is the appropriate change which
        // works for both compilers.
    };
};
```

The following code would result in the error `CCN5413:"A::B" is already declared with a different access:`

```
class A {
  public:
    class B;
    const B& foo();
  private:
    class B {};
};
```

This can be solved by either moving the definition of Class B to the public part of class A (before the declaration of `foo()`) or moving the declaration of the member function foo to the private of class A (after the class B definition).

typedefs

This code will generate error `CCN5193: A typedef name cannot be used in this context`. Do not use the typedef-name; instead, use the name of the class:

```
class A { };
typedef A B;
class C {
    friend class B; // Should be friend class A;
};
```

Overloading Ambiguities

There are now floating point and long double overloads of the standard math functions. For example, the following code, which would generate no errors in the OS/390 V2R10 compiler and earlier compilers, will produce under z/OS V1R2 the error message `CCN5219: The call to "pow" has no best match`.

```
#include <math.h>
int main()
{
    float a = 137;
```

```
    float b;
    b = pow(a, 2.0);        // The call to "pow" has no best match.
    return 0;
}
```

The solution is to cast `pow`'s arguments, or use the `LANGLVL(OLDMATH)` option, which removes the float and long double overloads. In this example casting `2.0` to be of type `float` solves the problem:

```
b = pow(a, (float)2.0);
```

The following generates a number of errors:

```
//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;
}
```

```
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 ".
```

Solutions include (depending on your access to classes A, B, and C):

- 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()

Syntax Errors with new

The OS/390 V2R10 compiler and earlier compilers treated the following two statements as semantically equivalent:

```
new (int *) [1];
new int* [1];
```

The first statement is syntactically incorrect even in older versions of the C++ Standard. However, previous versions of C++ accepted it. This inconsistency with the language standard was corrected for the z/OS V1R2 compiler; the first statement will now produce a compilation error.

Common Template Problems

If your code makes use of templates, you will be affected by various changes to the way the z/OS V1R2 compiler handles templates.

Changes in Name Resolution

The OS/390 V2R10 C++ compiler and earlier compilers do not parse or otherwise process a class or function template definition until it has been determined that an instantiation of that template is required. Template definitions for which no instantiation is required are never parsed by the OS/390 V2R10 C++ compiler and earlier compilers. By default, the z/OS V1R2 C++ compiler processes class and function template definitions in two phases:

- When the template definition is encountered by the compiler, the definition is parsed. Names that are used in the template definition and that are not dependent on the template parameters are resolved at this time.
- When it is determined that a specific instantiation of the template is required, names that are dependent on the template parameters are resolved and an implicit specialization is instantiated.

To approximate the behavior of the OS/390 V2R10 C++ compiler and earlier compilers, users of the z/OS V1R2 C++ compiler may use the `TMPLPARSE(NO)` option to override this default behavior. When the `TMPLPARSE(NO)` option is in effect, the first phase described above is delayed until it is determined that an instantiation is required. Template definitions for which no instantiation is required are not parsed. The `TMPLPARSE(NO)` option does not eliminate the distinction between the two phases.

An unqualified name that is not found by name lookup and not indicated to be a type by the `typename` keyword, is assumed to not name a type.

Unqualified name lookup does not consider template-dependent base classes.

As of z/OS V1R2, template-dependent base classes are not searched during name resolution. For example:

```
int *t=0;
template <class T> struct Base {
    U t;
};
template <class T> class C : public Base<T> {
    T f() {
        return t;  // refers to global int *t
    }
};
```

The keyword `typename` must be used to mark a qualified dependent name as a type. The following example illustrates this:

```
template <class T> struct A
{
    typedef int X;
};
template <class T> struct B:A <T>
{
    T::Y b1;      //error Y is not a type
    A <T>::X b2;  // error X is not a type
    void foo(X);  // error X is not a type
};
```

The errors can be fixed by changing the definition of B to:

```
template <class T> struct B : A <T>
{
    typename T::Y b1;
        // keyword "typename" tells parser Y is a type
    typename A<T>::X b2;
        // keyword "typename" tells parser X is a type
    void foo(typename A<T>::X);
        // keyword "typename" tells parser X is a type
};
```

template Keyword

As of z/OS V1R2, the `template` keyword is used to indicate templates in qualifiers. For example:

```
struct A {
    Template<class T> T f(T t) { return t;}
};
template <class T> class C {
    void g(T* a) {
    // The following would become ambiguous without
    // the keyword template
    int i = a->template f<int>(1);
}
C<A> c;
```

Template Specialization

As of z/OS V1R2, template specializations must be preceded with the string `template<>`. For example:

```
template <class T> class C {};
template <> C<int> { int i; };
```

Explicit Call to Destructor of Scalar Type

This problem is not template-specific, but usually occurs in templates. For example:

```
typedef int INT;
INT *p;
// ...
p->INT::~INT(); // ok in z/OS V1R2 C++
```

The OS/390 V2R10 compiler and earlier compilers give a warning to the explicit destructor call. You can ignore this warning.

Changes to friend Declarations

In z/OS V1R2, `friend` declarations in templates may not have the same meaning as with earlier compilers. For example, the following code will generate a warning message:

```
struct A {} a;
template <class T> struct S;
template <class T> void f(T&, S<T>&) {}
template <class T> A& operator << (A&, S<T>&) { return a; }
template <class T> struct S
{
    friend void f (T&, S&); // no explicit arguments
    friend A& operator <<(A&, S&); // no explicit arguments
};
```

To migrate this code, the `friend` declarations should be changed to include explicit template arguments:

```
template <class T> struct S
{
    friend void f<T> (T&, S&); // explicit argument T
    friend A& operator << <T>(A&, S&); // explicit argument T
};
```

Without the explicit arguments, the `friend` declarations will introduce non-template functions `'f(int&, S&)'` and `'operator <<(A&, S&)'` into global scope and these non-template functions (which have no corresponding definition) will be the friends of `S`.

With the template argument added explicitly, an instantiation of `S`, such as `S<int>`, will make the template instantiations `f<int>(int&, S<int>&)` and `operator << <int> (A&, S<int>&)`, friends of `S`.

The OS/390 V2R10 compiler and earlier compilers would not accept explicit template arguments on `friend` declarations. If you wish to maintain compatibility with earlier compilers, the explicit template arguments should be added with the use of a macro.

Changes to friend Declarator

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 OS/390 V2R10 and earlier compilers, friend declarations introduce the name of a nominated friend function to the scope that encloses the class containing the friend declaration. As of OS/390 V2R10, friend declarations do not introduce the name of a nominated friend function to the scope that encloses the class containing the friend declaration.

In the example source file below, the function name `lib_func1` is not known to the z/OS V1R2 C++ compiler at the point at which it is called in the function f. This source file will not compile successfully.

```
// g.C
// ---
class A {
    friend int lib_func1(int); // This function is from a library.
};
int f(){
    return lib_func1(1);
}
```

The example 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);
```

Length of External Variable Names

The z/OS V1R2 binder imposes a limit of 1024 characters for the length of external names. Both the OS/390 C++ compiler and z/OS C++ compiler may sometimes generate mangled names that are longer than this limit. This could occur more often when using the Standard Template Library with the z/OS V1R2 C++ compiler. Should you encounter this problem:

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

In a future release of the Program Management binder, IBM will raise the limit to match that of the prelinker (32K-1 characters).

Part 6. Appendixes

Appendix. Class Library Migration Considerations

This appendix provides some background information on the types of class libraries that are available with the C++ for MVS/ESA V3 and z/OS C/C++ compilers.

The following four class libraries are available for use with the z/OS C++ compilers, beginning with the C++ for MVS/ESA V3R1M0 compiler:

- I/O Stream Class Library
- Complex Mathematics Class Library
- Application Support Class Library
- Collection Class Library

The C++ for MVS/ESA V3R2 compiler introduced the Database Access Class Library Utility, which was removed in OS/390 V2R4.

All libraries are available in both statically bindable and DLL forms.

In previous releases, the Application Support Class library and the Collection Class library were also offered in SOM versions. Starting with OS/390 V2R10, these SOM-enabled class libraries have been removed.

In C++ class libraries, references to methods are dependent upon the order of the method entries in a virtual function table. When new methods are added to a library, the order of the methods can change, and therefore existing applications using those methods may no longer work. Between releases or modification levels, migration impacts may also occur if there are changes to the interfaces or semantics of existing functions within a class library.

Whether an application is statically bound to or uses the DLL form of a class library will also determine whether or not there are executable incompatibilities. Statically-bound applications do not usually encounter release-to-release executable incompatibilities unless they are recompiled/relinked from source or relinked from objects with the new release. Applications that use the DLL form, however, may encounter release-to-release executable incompatibilities. Source and object incompatibilities may occur regardless of whether an application is statically bound or uses the DLL.

For more information on the topics mentioned above, refer to the following:

- *IBM Open Class Library User's Guide*
- *IBM Open Class Library Reference*
- The "Building and Using Dynamic Link Libraries" chapter in the *z/OS C/C++ Programming Guide* (for information on dynamic linking)
- *z/OS C/C++ User's Guide* (for information on static linking)

Notices

This information was developed for products and services offered in the U.S.A.

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 (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, 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 OR CONDITIONS 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:

Lab Director
IBM Canada Ltd. Laboratory
B3/KB7/8200/MKM
8200 Warden Avenue
Markham, Ontario L6G 1C7
Canada

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.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples may include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on the z/OS 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. 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 IBM's application programming interfaces.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Programming Interface Information

This publication documents *intended* Programming Interfaces that allow the customer to write z/OS C/C++ programs.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States or other countries or both:

AD/Cycle	AIX	C/370
C/MVS	C++/MVS	CICS
IBM	IMS	Language Environment
MVS	MVS/ESA	Open Class

OpenEdition	OS/390	S/370
S/390	SAA	SOM
SP	System/370	System Object Model
VisualAge	z/OS	

UNIX is a registered trademark of The Open Group in the U.S. and/or other countries.

Other company, product, and service names may be trademarks or service marks of others.

Bibliography

This bibliography lists the publications for IBM products that are related to the z/OS C/C++ product. 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 C/C++ users. Refer to *z/OS Information Roadmap*, SA22-7500, for a complete list of publications belonging to the z/OS product.

Related publications not listed in this section can be found on the *IBM Online Library Omnibus Edition MVS Collection*, SK2T-0710, the *z/OS Collection*, SK3T-4269, or on a tape available with z/OS.

z/OS

- *z/OS Introduction and Release Guide*, GA22-7502
- *z/OS Planning for Installation*, GA22-7504
- *z/OS Summary of Message Changes*, SA22-7505
- *z/OS Information Roadmap*, SA22-7500

z/OS C/C++

- *z/OS C/C++ Programming Guide*, SC09-4765
- *z/OS C/C++ User's Guide*, SC09-4767
- *C/C++ Language Reference*, SC09-4815
- *z/OS C/C++ Messages*, GC09-4819
- *z/OS C/C++ Run-Time Library Reference*, SA22-7821
- *z/OS C Curses*, SA22-7820
- *z/OS C/C++ Compiler and Run-Time Migration Guide*, GC09-4913
- *IBM Open Class Library User's Guide*, SC09-4811
- *IBM Open Class Library Reference*, SC09-4812
- *Debug Tool User's Guide and Reference*, SC09-2137
- *Standard C++ Library Reference*, which is available at: `http://www.ibm.com/software/ad/c390/czos/czosdocs.html`

z/OS Language Environment

- *z/OS Language Environment Concepts Guide*, SA22-7567
- *z/OS Language Environment Customization*, SA22-7564
- *z/OS Language Environment Debugging Guide*, GA22-7560
- *z/OS Language Environment Programming Guide*, SA22-7561
- *z/OS Language Environment Programming Reference*, SA22-7562
- *z/OS Language Environment Run-Time Migration Guide*, GA22-7565
- *z/OS Language Environment Writing Interlanguage Applications*, SA22-7563
- *z/OS Language Environment Run-Time Messages*, SA22-7566

Assembler

- *HLASM Language Reference*, SC26-4940
- *HLASM Programmer's Guide*, SC26-4941

COBOL

- *COBOL for OS/390 & VM Compiler and Run-Time Migration Guide*, GC26-4764
- *COBOL for OS/390 & VM Programming Guide*, SC26-9049
- *COBOL for OS/390 & VM Language Reference*, SC26-9046
- *COBOL for OS/390 & VM Diagnosis Guide*, GC26-9047
- *COBOL for OS/390 & VM Licensed Program Specifications*, GC26-9044
- *COBOL for OS/390 & VM Customization under OS/390*, GC26-9045
- *COBOL Millenium Language Extensions Guide*, GC26-9266

PL/I

- *VisualAge PL/I Language Reference*, SC26-9476
- *PL/I for MVS & VM Language Reference*, SC26-3114
- *PL/I for MVS & VM Programming Guide*, SC26-3113
- *PL/I for MVS & VM Compiler and Run-Time Migration Guide*, SC26-3118

VS FORTRAN

- *Language and Library Reference*, SC26-4221
- *Programming Guide*, SC26-4222

CICS

- *CICS Application Programming Guide*, SC34-5702
- *CICS Application Programming Reference*, SC34-5703
- *CICS Distributed Transaction Programming Guide*, SC34-5708
- *CICS Front End Programming Interface User's Guide*, SC34-5710
- *CICS Messages and Codes*, GC33-5716
- *CICS Resource Definition Guide*, SC34-5722
- *CICS System Definition Guide*, SC34-5725
- *CICS System Programming Reference*, SC34-5726
- *CICS User's Handbook*, SX33-6116
- *CICS Family: Client/Server Programming*, SC34-1435
- *CICS Transaction Server for OS/390 Migration Guide*, GC34-5699
- *CICS Transaction Server for OS/390 Release Guide*, GC34-5701
- *CICS Transaction Server for OS/390: Planning for Installation*, GC34-5700

DB2

- *DB2 Administration Guide*, SC26-9931
- *DB2 Application Programming and SQL Guide*, SC26-9933
- *DB2 ODBC Guide and Reference*, GC26-9941
- *DB2 Command Reference*, SC26-9934
- *DB2 Data Sharing: Planning and Administration*, SC26-9935
- *DB2 Installation Guide*, GC26-9936
- *DB2 Messages and Codes*, GC26-9940
- *DB2 Reference for Remote DRDA Requesters and Servers*, SC26-9942
- *DB2 SQL Reference*, SC26-9944

- *DB2 Utility Guide and Reference*, SC26-9945

IMS/ESA

- *IMS/ESA Application Programming: Design Guide*, SC26-8728
- *IMS/ESA Application Programming: Transaction Manager*, SC26-8729
- *IMS/ESA Application Programming: Database Manager*, SC26-8727
- *IMS/ESA Application Programming: EXEC DLI Commands for CICS and IMS*, SC26-8726

QMF

- *Introducing QMF*, GC26-9576
- *Using QMF*, SC26-9578
- *Developing QMF Applications*, SC26-9579
- *Reference*, SC26-9577
- *Installing and Managing QMF on MVS*, SC26-9575
- *Messages and Codes*, SC26-9580

DFSMS

- *z/OS DFSMS Introduction*, SC26-7397
- *z/OS DFSMS: Managing Catalogs*, SC26-7409
- *z/OS DFSMS: Using Data Sets*, SC26-7410
- *z/OS DFSMS Macro Instructions for Data Sets*, SC26-7408
- *z/OS DFSMS Access Method Services*, SC26-7394
- *z/OS DFSMS Program Management*, SC27-1130

INDEX

Special Characters

A

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

Program Number: 5694–A01

Printed in the United States of America

GC09-4913-00