C/C++

C/C++ for z/VM User's Guide

SC09-7625-00

C/C++

C/C++ for z/VM User's Guide

SC09-7625-00

Note

Before using this information and the product it supports, be sure to read the general information under “Notices” on page 133.

First Edition (June 16, 2003)

This edition, SC09-7625-00, applies to C/C++ for z/VM Version 1, Release 1.0, program number 5654-A22, and to all subsequent releases of this product until otherwise indicated in new editions.

Contents

What is in this document

Part One of this document is the **Introduction**. This part:

- Introduces the C and C++ languages and the C/C++ for z/VM product and documentation.
- Describes differences between z/OS C/C++ and C/C++ for z/VM.
- Lists some issues that may arise when migrating from other C and C++ standards and compilers.

Part Two of this document is the **User's Reference**. This part reviews the basic steps for compiling, binding and running C/C++ for z/VM programs and describes some considerations for using the compiler under the VM/CMS operating system. It also describes the options available to you at compile, bind and run-time.

Part Three of this document is **Compiling, Binding and Running**. This part describes how to compile, bind, and run a C/C++ program using the Language Environment under VM/CMS.

Part Four of this document is **z/VM OpenExtensions: Compiling, Binding and Running**. This part describes how to compile, bind, and run a C/C++ for z/VM program under z/VM OpenExtensions.

Part Five of this document is **C/C++ for z/VM Utilities**. This section contains information about the C/C++ for z/VM utilities that you can use under VM/CMS.

Part Six of this document is **z/VM OpenExtensions: Utilities**. This part contains information about the z/VM OpenExtensions utilities.

The **Appendixes** offer advanced reference material including error messages and return codes.

The Glossary of terms, Bibliography, and Index sections are available for your reference.

Chapter 1. Introduction

Notes about the C/C++ for z/VM product

The C/C++ for z/VM licensed program is the language-centered C/C++ application development environment on the z/VM platform.

C/C++ for z/VM includes a C/C++ compiler component (referred to as the C/C++ for z/VM compiler) and some C/C++ application development utilities.

About this document

This edition of the *C/C++ for z/VM User's Guide* is intended for users of the C/C++ for z/VM with Language Environment. C/C++ for z/VM is a VM enabled version of the Version 1 Release 2 Modification 0 z/OS C/C++ compiler. It provides you with information about implementing (compiling, binding, and running) programs written in C and C++. It contains guidelines for preparing C and C++ programs to run under the z/VM operating system.

To use this document, you must have a working knowledge of the C and C++ programming languages, the operating system, and where appropriate, the related products. This includes the Language Environment product and z/VM OpenExtensions.

The document includes information about the use of C/C++ for z/VM and utilities that are provided with the compiler and library products, to compile, bind, and run C and C++ applications.

Terminology used in this document

C/C++ for z/VM is used as an abbreviation of C/C++ for z/VM.

Throughout this document, the term *filename* is used to refer to both files in general, regardless of the specific file system in which they reside, and also more specifically to refer to the name component of a minidisk or shared file system file identifier. The intended usage should be clear from context.

It is often necessary, however, to make a distinction between files that reside in the byte file system, and those that reside on minidisks or in the shared file system (but not in the byte file system). For convenience, the former will be referred to as "BFS files", and the latter as "CMS files".

The term ddname is used to refer to a data definition name. The relation of a ddname to one or more CMS files is achieved by using the FILEDEF command or for a BFS file by using the OPENVM PATHDEF CREATE command.

The term FILEDEF is used to refer to the data definition created by the use of the FILEDEF command.

The term PATHDEF is used to refer to the data definition created by the use of the OPENVM PATHDEF CREATE command.

The term program module is defined as the output of the binder. A collective term for program object and load module.

The C language

The C language is a general purpose, function-oriented programming language that allows a programmer to create applications quickly and easily. C provides high-level control statements and data types as do other structured programming languages, and it also provides many of the benefits of a low-level language.

The C++ language

The C++ language is based on the C language, but incorporates support for object-oriented concepts. For a detailed description of the differences between C++ and C, refer to the *C/C++ Language Reference*.

The C++ language introduces classes, which are user-defined data types that may contain data definitions and function definitions. You can use classes from established class libraries, develop your own classes, or derive new classes from existing classes by adding data descriptions and functions. New classes can inherit properties from one or more classes. Not only do classes describe the data types and functions available, but they can also hide (encapsulate) the implementation details from user programs. An object is an instance of a class.

The C++ language also provides templates and other features that include access control to data and functions, and better type checking and exception handling. It also supports polymorphism and the overloading of operators.

Common features of the C and C++ compilers

The C and C++ compilers offer many features to help your work:

- Optimization support.

  The OPTIMIZE compiler option instructs the compiler to optimize the machine instructions it generates to produce faster-running object code to improve application performance at run-time.
- DLLs (dynamic link libraries) to share parts among applications or parts of applications , and dynamically link to exported variables and functions at Run-Time.

  DLLs allow a function reference or a variable reference in one executable to use a definition located in another executable at Run-Time. You can use both load-on-reference and load-on-demand DLLs. When your program refers to a function or variable which resides in a DLL, C/C++ for z/VM generates code to load the DLL and access the functions and variables within it. This is called load-on-reference. Alternatively, your program can use C library functions to load a DLL and look up the address of functions and variables within it. This is called load-on-demand. Your application code explicitly controls load-on-demand DLLs at the source level.

  You can use DLLs to split applications into smaller modules and improve system memory usage. DLLs also offer more flexibility for building, packaging, and redistributing applications.
- Full program reentrancy.
- Locale-based internationalization support derived from the IEEE POSIX 1003.2-1992 standard. Also derived from the X/Open CAE Specification, System Interface Definitions, Issue 4 and Issue 4 Version 2. This allows programmers to use locales to specify language/country characteristics for their applications.
- Year 2000 support
- Support for the Euro currency.

Class library

C/C++ for z/VM provides the following class library:

- C++ Standard Library, including the Standard Template Library (STL) and other library features of ISO C++ 1998

The C++ Standard Library includes the following:

- The C++ Standard I/O Stream Library for performing input and output (I/O) operations
- The C++ Standard Complex Mathematics Library for manipulating complex numbers
- The Standard Template Library (STL) which is composed of C++ template-based algorithms, container classes, iterators, localization objects, and the string class

Utilities

C/C++ for z/VM provides the following utilities:

- The CXXFILT utility to map C++ mangled names to the original source.
- The DSECT Conversion Utility to convert descriptive assembler DSECTs into C/C++ data structures.
- The localedef utility to read the locale definition file and produce a locale object that the locale-specific library functions can use.

Language Environment

C/C++ for z/VM exploits the runtime library and common library of base routines available with z/VM 4.4.0 and the C/C++ Language Environment for z/VM (referred to as Language Environment.) The C/C++ Language Environment for z/VM is based on z/OS V1 R4 Language Environment.

Language Environment establishes a common run-time environment and common run-time services for language products, user programs, and other products.

The common execution environment is made up of data items and services performed by library routines available to a particular application running in the environment. The services that Language Environment can provide to your application include:

- Services that satisfy basic requirements common to most applications. These include support for the initialization and termination of applications, allocation of storage, support for interlanguage communication (ILC) and condition handling.
- Extended services often needed by applications. These functions are contained within a library of callable routines, and include interfaces to operating system functions and a variety of other commonly used functions.
- Run-time options that help the execution, performance tuning, performance, and diagnosis of your application.
- Access to operating system services: z/VM OpenExtensions services are available to an application programmer or program through the C/C++ for z/VM language bindings.
- Access to language-specific library routines, such as the C/C++ for z/VM functions.

z/VM OpenExtensions

z/VM OpenExtensions provides capabilities under z/VM to make it easier to implement or port applications in an open, distributed environment.

z/VM OpenExtensions Services

z/VM OpenExtensions Services are available to C/C++ for z/VM application programs through the C language bindings available with Language Environment or z/VM Version 4.4.

Together, the z/VM OpenExtensions Services, Language Environment, and C/C++ for z/VM provide an application programming interface that supports industry standards.

Support for POSIX and UNIX like interfaces includes:

- OpenExtensions Services which include C-language programming interfaces defined by the IEEE POSIX 1003.1 standard (although the `fork()` function is only partially implemented) and subsets of the draft 1003.1a and 1003.1c standards, as well as z/VM OpenExtensions-unique extensions
- OpenExtensions Shell and Utilities feature, which provides a UNIX like user interface and an application development environment for creating C/C++ for z/VM programs for OpenExtensions, including the following utilities:

  c89 Compile and link-edit C/C++ for z/VM applications

  make Software build and maintenance tool

  ar Create and maintain library archives
- Byte File System (BFS), which provides the POSIX file system

This support offers:

- Program portability (with support for POSIX.1 and POSIX.1a) across multi-vendor operating systems
- A byte file system (BFS) in VM (with support for POSIX.1) including access to data in either CMS record files or the BFS
- A UNIX like user interface (with support for POSIX.2) including access to services provided through the OpenExtensions Shell and Utilities feature
- Application threads (with support for a subset of POSIX.1c)
- OpenExtensions extensions which provide VM-specific support beyond the defined standards.

This support is integrated with z/VM and Language Environment for use by both existing z/VM applications and for new OpenExtensions applications.

z/VM OpenExtensions provides capabilities under z/VM to make it easier to implement or port applications in an open, distributed environment.

Application developers familiar with UNIX like environments will find z/VM OpenExtensions Shell and Utilities a familiar C application development environment. Those familiar with existing z/VM development environments may find that the OpenExtensions environment can enhance their productivity. Refer to the *z/VM OpenExtensions User's Guide* for more information on the Shell and Utilities.

C/C++ for z/VM applications with OpenExtensions (POSIX) services

To make use of z/VM OpenExtensions Services, a C/C++ for z/VM program needs to be running on a z/VM 4.4.0 or later system. Furthermore, the program must be an OpenExtensions POSIX C/C++ for z/VM program with Language Environment run-time option POSIX(ON), or it must use the interoperability support for OpenExtensions. (See "C/C++ for z/VM applications with OpenExtensions interoperability" on page 5 for further information.)

A C/C++ for z/VM program can make use of OpenExtensions Services in one of the following ways:

- The program is invoked from another program, or from the OpenExtensions shell, using `spawn()` or one of the exec functions.
- The program is invoked using the POSIX `system()` call.
- The program is invoked from the CMS command line with the POSIX(ON) override option or through the OPENVM RUN command.

Functions with dependencies on the z/VM OpenExtensions kernel, such as `spawn()`, or the threading functions, such as `pthread_create()` are strictly limited to use within the OpenExtensions POSIX environment.

C/C++ for z/VM applications with OpenExtensions interoperability

OpenExtensions interoperability is used to describe the fact that:

- z/VM OpenExtensions applications running under POSIX(ON) can access traditional VM services and data.
- Traditional VM applications running under POSIX(OFF) can access the z/VM OpenExtensions Services that permit access to BFS data.

For example, for functions such as `fopen()` and `freopen()`, the following statement will open a BFS file named parts.instock:

```
fopen("./parts.instock","r")
```

The next statement will open a CMS record file named PARTS INSTOCK:

```
fopen("//parts.instock","r")
```

Changing the run-time option POSIX(OFF) to POSIX(ON) will affect the environment in which these functions execute. For example, the following statement will open the BFS file named, if POSIX(ON) is in effect, and will open the CMS record file if POSIX(OFF) is in effect:

```
fopen("parts.instock","r")
```

Some of the C language functions that use z/VM OpenExtensions Services can be invoked from applications running in the non-POSIX CMS environment, as specified with the run-time option POSIX(OFF).

For example, the following statement will open a BFS file named parts.instock whether the application is running under POSIX(OFF) or POSIX(ON):

```
open("parts.instock",O_RDONLY)
```

For more information on the C language functions available under the z/VM OpenExtensions environment that can be invoked by applications running in the non-POSIX VM environment, see the *C/C++ for z/VM Run-Time Library Reference*.

C/C++ for z/VM and related publications

- *C/C++ for z/VM Run-Time Library Reference*, SC09-7624
- *C/C++ for z/VM User's Guide*, SC09-7625
- *C/C++ Language Reference*, SC09-4815
- *z/OS C/C++ Messages*, GC09-4819
- *z/OS C/C++ User's Guide*, SC09-4767
- *z/OS C/C++ Compiler and Run-Time Migration Guide*, GC09-4913
- *z/OS Language Environment Debugging Guide*, GA22-7560

- *z/OS Language Environment Run-Time Messages*, SA22-7566
- *z/OS C/C++ Programming Guide*, SC09-4765
- *z/OS C Curses*, SA22-7820
- *z/OS C/C++ Run-Time Library Reference*, SA22-7821

Softcopy examples

Most of the larger examples in the following documents are available in machine-readable form:

- *C/C++ for z/VM User's Guide*, SC09-7625
- *C/C++ Language Reference*, SC09-4815
- *z/OS C/C++ Programming Guide*, SC09-4765

In the following documents, a label on an example indicates that the example is distributed in softcopy. The label is a filename on the C/C++ for z/VM product disk. The labels have the form `CCNxyyy` or `CLBxyyy`, where x refers to a publication:

- R and X refer to *C/C++ Language Reference*
- G refers to *z/OS C/C++ Programming Guide*
- U refers to *C/C++ for z/VM User's Guide*

Examples labelled as `CCNxyyy` appear in *C/C++ Language Reference*, *z/OS C/C++ Programming Guide*, and *C/C++ for z/VM User's Guide*. Examples labelled as `CLBxyyy` appear in the *C/C++ for z/VM User's Guide*.

Conventions

Throughout this document, the following conventions are used:

- *C/C++ for z/VM* is used to represent C/C++ for z/VM.
- *VM/CMS* is used to represent the CMS environment of the z/VM operating system and its subsequent releases.
- *z/VM OpenExtensions* is used to represent the OpenExtensions for z/VM environment.

How to read syntax diagrams

This document describes the syntax for commands, directives, and statements, using the following structure:

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

  A double right-arrowhead indicates the beginning of a command, directive, or statement. A single right-arrowhead indicates that it is continued on the next line. (In the following diagrams, "statement" represents a command, directive, or statement).

  ```
  ►►—statement——►
  ```

  The following indicates a continuation; the opposing arrowheads indicate the end of a command, directive, or statement.

  ```
  ►—statement—►◄
  ```

  Diagrams of syntactical units other than complete commands, directives, or statements look like this:

  ```
  ►—statement—►
  ```

- Required items are on the horizontal line (the main path).

- Optional items are below the main path.

- Default items are above the main path.

- If you can choose from two or more items, they are vertical in a stack.
  If you *must* choose one of the items, one item of the stack is on the main path.

  If choosing one of the items is optional, the entire stack is below the main path.

- An arrow that returns to the left above the main line indicates an item that you can repeat.

  A repeat arrow above a stack indicates that you can make more than one choice from the stacked items, or repeat a single choice.
- Keywords are not italicized, and should be entered exactly as shown. You must spell keywords exactly as shown in the syntax diagram. Variables are in italics (in hardcopy) and lowercase letters for example, *filename*). They represent user-supplied names or values.
- If the syntax diagram shows punctuation marks, parentheses, arithmetic operators, or other nonalphanumeric characters, you must enter them as part of the syntax.

Note: You do not always require the white space between tokens. You should, however, include at least one blank space between tokens unless otherwise specified.

Differences/Restrictions from z/OS C/C++

Facilities/Compiler Options Not Supported

The following z/OS C/C++ compiler features and their associated compiler options are not supported in C/C++ for z/VM:

- Interprocedural Analysis

  The associated compiler option is IPA.

- ASCII support

  The associated compiler option is ASCII.
- IBM Open Class Library
- Host Performance Analyzer

Restrictions

- TEMPINC compiler option

  Under z/OS, this option specifies, either specifically or by default, a location where the compiler will place all C++ template instantiation files. If the source file resides in a dataset, the default location is a PDS with a low-level qualifier of TEMPINC. If the source resides in the HFS, the default location is the HFS directory `./tempinc`. A PDS name or HFS directory name can be specified to override this.

  Under CMS, because of the way the compiler is implemented, the source file always appears to the compiler as residing in the BFS. Therefore, the default TEMPINC location is always the BFS. Also, this option will only work if all of the template declaration files (.h) and template definition files (.c) are in the BFS. The template instantiation files will then be written to the `./tempinc` directory by default or a directory specified on the TEMPINC option. A BFS directory is the only valid location on the TEMPINC option under CMS. That is, the location name must contain a '/', for example:

  ```
  TEMPINC(./fred)
  ```

  or

  ```
  TEMPINC(fred/jim)
  ```

  If the location is not a BFS directory, the compiler will issue a `CCN6001` message.

  Note: The TEMPLATEREGISTRY option is another option which manages template instantiation and has no restrictions under CMS. It also imposes no requirements on the organization of your source code, unlike TEMPINC. It is recommended therefore, that TEMPLATEREGISTRY be used in preference to TEMPINC.

C for VM/ESA Migration Considerations

Source Program Compatibility

In general, you can use source programs with the C/C++ for z/VM 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 for VM/ESA V3R1 compiler running with Language Environment V1R8.

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

Supporting the ISO Standard

C/C++ for z/VM supports the ISO 1998 standard. The following sections detail the differences between previous versions of C/C++ and the ISO 1998 standard.

Changes in Language Features:

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 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 C/C++ for z/VM, this means `::i` (the `i` declared in file scope).
- In earlier compilers, this means the local `i`. To maintain this context within C/C++ for z/VM, use the `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 C/C++ for z/VM, 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 C/C++ for z/VM 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 C/C++ for z/VM:

- 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 earlier compilers. There is no available option in C/C++ for z/VM to enable the old behavior.

Operator new and delete: As of C/C++ for z/VM, operator new throws `std::bad_alloc` if memory allocation fails.

In earlier compilers, operator new returned 0 to indicate allocation failure.

As of C/C++ for z/VM, 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: Earlier compilers do not support overloaded delete.

New Language Features:

New Keywords: As of C/C++ for z/VM, 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 earlier compilers. Code ported to 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 C/C++ for z/VM, you need to undefine the macro.

The bool Type: The `bool` type is not supported in earlier compilers. Be aware that relational operators return `bool` in C/C++ for z/VM, while it returns `int` in previous compilers.

To disable this keyword in C/C++ for z/VM, use the `NOKEYWORD(BOOL)` option.

mutable Keyword: As of C/C++ for z/VM, 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 earlier compilers. Code ported to 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 C/C++ for z/VM, you need to undefine the macro.

To disable this keyword in C/C++ for z/VM, use the `NOKEYWORD(MUTABLE)` option.

wchar_t as Simple Type: The C/C++ for z/VM compiler defines `wchar_t` as a simple type. Earlier compilers define it as a `typedef`.

explicit: 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 C/C++ for z/VM use the `NOKEYWORD(EXPLICIT)` option.

C++ cast: C/C++ for z/VM has new cast operators: `const_cast`, `dynamic_cast`, `reinterpret_cast` and `static_cast`. These were not supported in 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 C/C++ for z/VM, 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 earlier compilers, but which will produce errors or warnings in the C/C++ for z/VM 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 earlier compilers, will produce under C/C++ for z/VM 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: 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 C/C++ for z/VM 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 C/C++ for z/VM compiler handles templates.

Changes in Name Resolution: 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 earlier compilers. By default, the C/C++ for z/VM 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 earlier compilers, users of the C/C++ for z/VM 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 C/C++ for z/VM, 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 C/C++ for z/VM, 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 C/C++ for z/VM
```

Earlier compilers give a warning to the explicit destructor call. You can ignore this warning.

Changes to friend Declarations: In C/C++ for z/VM, `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`.

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 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 C/C++ for z/VM, 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 C/C++ for z/VM 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);
```

LANGLVL(ANSI)

If you specify `LANGLVL(ANSI)`, the compiler recognizes `char`, `unsigned char`, and `signed char` as three distinct types. This conforms to the ISO C++ 1998 standard.

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.

Long Long Data Type

As of C/C++ for z/VM, 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 C/C++ for z/VM, 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 C/C++ for z/VM When `LANGLVL(LONGLONG)` is turned on the `_LONG_LONG` macro is defined by the compiler.

Other Migration Considerations

This chapter provides additional considerations on migrating applications from the compilers previously listed.

Compiler Options

DECK Compiler Option: As of C/C++ for z/VM, the `DECK` compiler option is no longer supported. The replacement for `DECK` functionality is the `OBJECT` compiler option.

ENUM Compiler Option: C/C++ for z/VM 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).

HWOPTS Compiler Option: In IBM C for VM/ESA, the `HWOPTS` compiler option directed the compiler to generate code to take advantage of different hardware. As of C/C++ for z/VM, the `HWOPTS` compiler option is no longer supported. The replacement for it is the `ARCHITECTURE` option.

OPTIMIZE Compiler Option: In IBM C for VM/ESA:

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

With C/C++ for z/VM:

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

The levels of optimization for `OPT` for IBM C for VM/ESA and C/C++ for z/VM is significantly different. The underlying compiler technology within these compilers has changed significantly.

Memory Considerations

Memory requirements for compilation may increase for C/C++ for z/VM. If you cannot recompile an application that you successfully compiled with IBM C for VM/ESA, increase the virtual machine size.

Compiler Listings

As of C/C++ for z/VM, `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, refer to the *z/OS C/C++ User's Guide*.

Chapter 2. User's Reference

C Example

This chapter contains an example of the basic steps for compiling, binding, and running a C program.

If you have not yet compiled a C/C++ for z/VM program or read the other chapters in this book, some concepts in this chapter may be unfamiliar. This chapter outlines the steps to compile, bind, and run your program under VM/CMS. Refer to relevant sections of the book for clarification as you read the examples of compiling, binding and running.

Example of a C Program

The following example shows a simple program that converts temperatures in Celsius to Fahrenheit. You can either enter the temperatures on the command line or be prompted for the temperature.

In this example, the main program calls the function

```
convert
```

to perform the conversion of the Celsius temperature to a Fahrenheit temperature and to print the result.

CCNUAAM

```
#include <stdio.h>                  1

#include "ccnuaan.h"                2

void convert(double);               3

int main(int argc, char **argv)     4
{
    double c_temp;                  5

    if (argc == 1) {  /* get Celsius value from stdin */
       int ch;

       printf("Enter Celsius temperature: \n");    6

       if (scanf("%f", &c_temp) != 1) {
          printf("You must enter a valid temperature\n");
       }
       else {
          convert(c_temp);          7
       }
    }
    else {  /* convert the command-line arguments to Fahrenheit */
       int i;

       for (i = 1; i < argc; ++i) {
           if (sscanf(argvïiÙ, "%f", &c_temp) != 1)
              printf("%s is not a valid temperature\n",argvïiÙ);
           else
              convert(c_temp);      7
       }
    }
}

void convert(double c_temp) {       8
   double f_temp = (c_temp * CONV + OFFSET);
   printf("%5.2f Celsius is %5.2f Fahrenheit\n",c_temp, f_temp);
}
```

Figure 1. Celsius to Fahrenheit Conversion

CCNUAAN

```
/*************************************************************
 *  User include file:  ccnuaan.h                            *   9
 *************************************************************/

#define CONV   (9./5.)
#define OFFSET 32
```

Figure 2. User #include File for Conversion Program

1 This preprocessor directive includes the system file that contains the declarations of standard library functions, such as the `printf()` function used by this program.

The compiler searches for the file named STDIO H or for the member STDIO of the VM/CMS MACLIBs, depending on the options that are set.

For a description of the file modes used in the search, see “Search Sequences for Include Files” on page 67.

2 This preprocessor directive includes a user file that defines constants that are used by the program.

The compiler searches for a file called `CCNUAAN`. See “Search Sequences for Include Files” on page 67 for a description of the file modes used in the search.

If the compiler cannot locate the file in the user libraries, the system libraries are searched.

3 This is a function prototype declaration. This statement declares `convert()` as an external function having one parameter.

4 The program begins execution at this entry point.

5 This is the automatic (local) data definition to `main()`.

6 This `printf()` statement is a call to a C library function that allows you to format your output and print it on the standard output device. The `printf()` function is declared in the C standard I/O header file `stdio h` included at the beginning of the program.

7 This statement contains a call to the function `convert()`. It was declared earlier in the program as receiving one double value, and not returning a value.

8 This is a function definition. In this example, the declaration for this function appears immediately before the definition of the `main()` function. The C code for the function is in the same file as the code for the `main()` function.

9 This is the user include file containing the definitions for `CONV` and `OFFSET`

If you need more details on the constructs of the C language, see the *C/C++ Language Reference* and the *C/C++ for z/VM Run-Time Library Reference*.

Compiling, Binding, and Running the C Example

In general, you can compile, bind, and run C programs under CMS or the z/VM OpenExtensions shell. For more information, see “Compiling” on page 55 and “Binding and Running” on page 69.

Non-XPLINK and XPLINK Under CMS

If the sample C program (CCNUAAM) was stored in `CCNUAAM C L`, and the sample include file (CCNUAAN) was stored in `CCNUAAN H L`, the following set of commands would compile, bind, and run the source code, using the Language Environment:

```
GLOBAL LOADLIB SCEERUN              1
CC CCNUAAM C L                      2
-- or, for XPLINK --
CC CCNUAAM C L (XPLINK              2
CMOD CCNUAAM                        3
-- or, for XPLINK --
CMOD CCNUAAM (XPLINK                3
GLOBAL LOADLIB SCEERUN              4
CCNUAAM                             5
```

Figure 3. Commands to Compile, Bind, and Run a C Program under VM/CMS

1 Makes the library available to the compiler.

2 Compiles CCNUAAM C L and stores the object module in CCNUAAM TEXT A.

3 Using CCNUAAM TEXT A, created by the CC EXEC, generates an executable module called CCNUAAM MODULE using default options.

4 Makes the run-time library available to the exectuable module.

5 Runs CCNUAAM MODULE A using default options.

Non-XPLINK and XPLINK Under the z/VM OpenExtensions Shell

If the sample C program (CCNUAAM) was stored in ./ccnuaam.c, and the sample include file (CCNUAAN) was stored in ./ccnuaan.h, the following set of commands would compile, bind, and run the source code, using the Language Environment:

Note: In this example, the current working directory is used, so make sure that you are in the directory you want to use. Use the `pwd` command to display the current working directory, the `mkdir` command to create a new directory, and the `cd` command to change directories.

```
c89 -o conv ccnuaam.c                                    1
-- or, for XPLINK --
c89 -o conv -Wc,xplink -Wb,xplink ccnuaam.c              1
cms global loadlib sceerun                               2
conv                                                     3
```

Figure 4. Commands to Compile, Bind, and Run a C Program under z/VM OpenExtensions

1 Compiles and binds ccnuaam.c, and generates an executable module called `conv`.

2 Makes the run-time library available to the exectuable module.

3 Runs `conv` using default options.

C++ Examples

This chapter contains an example of the basic steps for compiling, binding, and running a C++ program.

If you have not yet compiled a C/C++ for z/VM program or read the other chapters in this book, some concepts in this chapter may be unfamiliar. This chapter outlines the steps to compile, bind, and run your program under VM/CMS. Refer to relevant sections of the book for clarification as you read the examples of compiling, linking and running.

Example of a C++ Program

The following example shows a simple C++ program that prompts you to enter a birth date. The program output is the corresponding biorhythm chart.

The program is written in object-oriented fashion. A class that is called `BioRhythm` is defined. It contains an object `birthDate` of class `BirthDate`, which is derived from the class `Date`. An object that is called `bio` of the class `BioRhythm` is declared.

The example contains 2 files. File `CCNUBRH` contains the classes that are used in the main program. File `CCNUBRC` contains the remaining source code.

If you need more details on the constructs of the C++ language, see the *C/C++ Language Reference* or the *C/C++ for z/VM Run-Time Library Reference*.

CCNUBRH

```
//
// Sample Program: Biorhythm
// Description   : Calculates biorhythm based on the current
//                 system date and birth date entered
//
// File 1 of 2-other file is CCNUBRC

using namespace std;

class Date {
  public:
    Date();
    int DaysSince(const char *date);

  protected:
    int curYear, curDay;
    static const int dateLen;
    static const int numMonths;
    static const int numDays[];
};
const int Date::dateLen = 10;
const int Date::numMonths = 12;
const int Date::numDays[Date::numMonths] = {
  31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
};
class BirthDate : public Date {
  public:
    BirthDate();
    BirthDate(const char *birthText);
    int DaysOld() { return(DaysSince(text)); }

  private:
    char text[dateLen+1];
};
```

Figure 5. Header File for the Biorhythm Example (Part 1 of 2)

```
class BioRhythm {
  friend ostream& operator<<(ostream&, BioRhythm&);

  public:
    BioRhythm(char *birthText) : birthDate(birthText) {
      age = birthDate.DaysOld();
    }
    BioRhythm() : birthDate() {
      age = birthDate.DaysOld();
    }
    ~BioRhythm() {}

    int AgeInDays() {
      return(age);
    }
    double Physical()  {
      return(Cycle(pCycle));
    }
    double Emotional() {
      return(Cycle(eCycle));
    }
    double Intellectual() {
      return(Cycle(iCycle));
    }
    int ok() {
      return(age >= 0);
    }

  private:
    int age;
    double Cycle(int phase) {
      return(sin(fmod((double)age, (double)phase) / phase * M_2PI));
    }
    BirthDate birthDate;
    static const int pCycle;
    static const int eCycle;
    static const int iCycle;
};

const int BioRhythm::pCycle=23;     // Physical cycle - 23 days
const int BioRhythm::eCycle=28;     // Emotional cycle - 28 days
const int BioRhythm::iCycle=33;     // Intellectual cycle - 33 days

ostream& operator<<(ostream&,BioRhythm&);
```

Figure 5. Header File for the Biorhythm Example (Part 2 of 2)

CCNUBRC

```
//
// Sample Program: Biorhythm
// Description   : Calculates biorhythm based on the current
//                 system date and birth date entered
//
// File 2 of 2-other file is CCNUBRH

#include <stdio.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include <iostream>
#include <iomanip>
#include "ccnubrh.h"  //BioRhythm class and Date class

using namespace std;
int main(void) {
  BioRhythm bio;
  int code;

  if (!bio.ok()) {
    cerr << "Error in birthdate specification - format is yyyy/mm/dd";
    code = 8;
  }
  else {
    cout << bio;  // write out birthdate for bio
    code = 0;
  }
  return(code);
}

ostream& operator<<(ostream& os, BioRhythm& bio) {
  os << "Total Days  : " << bio.AgeInDays() << "\n";
  os << "Physical    : " << bio.Physical() << "\n";
  os << "Emotional   : " << bio.Emotional() << "\n";
  os << "Intellectual: " << bio.Intellectual() << "\n";

return(os);
}

Date::Date() {
  time_t lTime;
  struct tm *newTime;

  time(&lTime);
  newTime = localtime(&lTime);
  cout << "local time is " << asctime(newTime) << endl;

  curYear = newTime->tm_year + 1900;
  curDay  = newTime->tm_yday + 1;
}
```

Figure 6. z/OS C++ Biorhythm Example Program (Part 1 of 2)

```
BirthDate::BirthDate(const char *birthText) {
  strcpy(text, birthText);
}

BirthDate::BirthDate() {
  cout << "Please enter your birthdate in the form yyyy/mm/dd\n";
  cin >> setw(dateLen+1) >> text;
}

Date::DaysSince(const char *text) {
  int year, month, day, totDays, delim;
  int daysInYear = 0;
  int i;
  int leap = 0;

  int rc = sscanf(text, "%4d%c%2d%c%2d",
                  &year, &delim, &month, &delim, &day);
  --month;
  if (rc != 5 || year  < 0 || year  > 9999 ||
                 month < 0 || month >   11 ||
                 day   < 1 || day   >   31 ||
                (day   > numDays[month]&& month != 1)) {
    return(-1);
  }
  if ((year % 4 == 0 && year % 100 != 0) || year % 400 == 0)
     leap = 1;

  if (month == 1 && day > numDays[month]) {
    if (day > 29)
      return(-1);
    else if (!leap)
      return (-1);
  }

  for (i=0;i<month;++i) {
    daysInYear += numDays[i];
  }
  daysInYear += day;

  // correct for leap year
  if (leap == 1 &&
     (month > 1 || (month == 1 && day == 29)))
     ++daysInYear;

  totDays = (curDay - daysInYear) + (curYear - year)*365;

  // now, correct for leap year
  for (i=year+1; i < curYear; ++i) {
    if ((i % 4 == 0 && i % 100 != 0) || i % 400 == 0) {
      ++totDays;
    }
  }
  return(totDays);
}
```

Figure 6. z/OS C++ Biorhythm Example Program (Part 2 of 2)

Compiling, Binding, and Running the C++ Example

In general, you can compile, bind, and run C++ programs under CMS or the z/VM OpenExtensions shell. For more information, see “Compiling” on page 55 and “Binding and Running” on page 69.

Non-XPLINK and XPLINK Under CMS

If the sample C++ program (CCNUBRC) was stored in CCNUBRC CPP L, and the sample include file (CCNUBRH) was stored in CCNUBRH H L, the following set of commands would compile, bind, and run the source code, using the Language Environment:

```
GLOBAL LOADLIB SCEERUN              1
CC CCNUBRC CPP L                    2
-- or, for XPLINK --
CC CCNUBRC CPP L (XPLINK            2
CMOD CCNUBRC (C++                   3
-- or, for XPLINK --
CMOD CCNUBRC (XPLINK C++            3
GLOBAL LOADLIB SCEERUN              4
CCNUBRC                             5
```

Figure 7. Commands to Compile, Bind, and Run a C++ Program under CMS

1 Makes the run-time library available to the compiler.

2 Compiles CCNUBRC C L and stores the object module in CCNUBRC TEXT A.

3 Using CCNUBRC TEXT A, created by the CC EXEC, generates an executable module called CCNUBRC MODULE using default options.

4 Makes the run-time library available to the exectuable module.

5 Runs CCNUAAM MODULE A using default options.

Non-XPLINK and XPLINK Under the z/VM OpenExtensions Shell

If the sample C++ program (CCNUBRC) was stored in ./ccnubrc.cpp, and the sample include file (CCNUBRH) was stored in ./ccnubrh.h, the following set of commands would compile, bind, and run the source code, using the Language Environment:

Note: In this example, the current working directory is used, so make sure that you are in the directory you want to use. Use the `pwd` command to display the current working directory, the `mkdir` command to create a new directory, and the `cd` command to change directories.

```
cxx -o bio ccnubrc.cpp                          1
-- or, for XPLINK --
cxx -o bio -Wc,xplink -Wb,xplink ccnubrc.cpp    1
cms global loadlib sceerun                      2
bio                                             3
```

Figure 8. Commands to Compile, Bind, and Run a C++ Program under z/VM OpenExtensions

1 Compiles and binds ccnubrc.cpp, and generates an executable module called `bio`.

2 Makes the run-time library available to the exectuable module.

3 Runs `bio` using default options.

Example of a C++ Template Program

A class template or generic class is a blueprint that describes how members of a set of related classes are constructed.

The following example shows a simple C++ program that uses templates to perform simple operations on linked lists. This program consists of ten files that are described and illustrated below.

The main program, CCNUTMP.CPP (see “CCNUTMP.CPP” on page 33) has two class templates: `List` (in the file CCNULST.C that uses CCNULST.H) and `Iterator` (in the file CCNUITR.C that uses CCNUITR.H). `List` is a template of a linked list, and `Iterator` is a template that walks a `List` class.

CCNULST.C

```
#include "ccnulst.h"
 template <class Item> void List<Item>::append(Item item) {
   GetNode();
   cur->node = item;
 }

 template <class Item> void List<Item>::GetNode() {
   if (cur) {
     cur->next = new Node<Item>;
     cur = cur->next;
   }
   else {
     cur = new Node<Item>;
     head = cur;
   }
   cur->next = 0;
   return;
 }
```

Figure 9. Template of a Linked List

CCNULST.H

```
#ifndef __CBCLIST_H
  #define __CBCLIST_H    1

  template <class Item> struct Node {
    Item node;
    struct Node<Item> *next;
  };

  template <class Item> class List {
    public:

      List() :cur(0), head(0) {}

        ~List() {}
      void append(Item item);
      Node<Item> *cur, *head;

    private:
      void GetNode();
  };
#ifndef __TEMPINC__
#include "ccnulst.c"
#endif
#endif
```

Figure 10. Header file for CCNULST.C

CCNUITR.C

```
#include "ccnuitr.h"
 template <class Item> Item& Iterator<Item>::operator++() {
   node = cur->node;
   cur  = cur->next;
   return(node);
 }

 template <class Item> int Iterator<Item>::eol() {
   return(cur == 0);
 }

 template <class Item> void Iterator<Item>::reset() {
   cur = head;
 }
```

Figure 11. Template of an Iterator

CCNUITR.H

```
#ifndef __CBCITER_H
 #define __CBCITER_H       1
 #include "ccnulst.h"
 template <class Item> class Iterator {
   public:
     Iterator(List<Item>& list)
             :cur(list.head), head(list.head) {}
     Item& operator++();
     int eol();
     void reset();

   private:
     Node<Item> *cur;
     Node<Item> *head;
     Item node;
 };
#ifndef __TEMPINC__
#include "ccnuitr.c"
#endif
#endif
```

Figure 12. Header file for CCNUITR.C

There are two template functions, `max(T,T)` (in the file CCNUMAX.C which uses CCNUMAX.H), and `min(T,T)` (in the file CCNUMAX.C which uses CCNUMIN.H). `max(T,T)` returns the maximum object of two objects, and `min(T,T)` returns the minimum object of two objects.

CCNUMAX.H

```
 template <class T> T& max(T& a, T& b);
#ifndef __TEMPINC__
#include "ccnumax.c"
#endif
```

CCNUMAX.C

```
template <class T> T& max(T& a, T& b) {
  if (a > b) return(a);
  else return(b);
}
```

CCNUMIN.H

```
 template <class T> T& min(T& a, T& b);
#ifndef __TEMPINC__
#include "ccnumin.c"
#endif
```

CCNUMIN.C

```
template <class T> T& min(T& a, T& b) {
  if (a < b) return(a);
  else return(b);
}
```

CCNUSTR.H

There is one simple class, `String`, defined in the file CCNUSTR.H.

```
#ifndef __CBCSTR_H
  #define __CBCSTR_H       1
  #include <string.h>
  class String {
    public:
      String() {
        str = new char[1];
        str[0]= '\0';
      }
      String(const char *s) {
        const int len = strlen(s);
        str = new char[len+1];
        memcpy(str, s, len+1);
      }

        ~String() {
        delete str;
      }
      void replace(const char *s) {
        const int len = strlen(s);
        char *newStr = new char[len+1];
        delete str;
        str = newStr;
        memcpy(str, s, len+1);
      }
      int operator >(String& rhs) {
        return(strcmp(str, rhs.string()));
      }
      int operator <(String& rhs) {
        return(!strcmp(str, rhs.string()));
      }
      const char *string() {
        return(str);
      }
    private:
      char *str;
  };
#endif
```

Figure 13. Definition of the String Class

CCNUTMP.CPP

```
#include "ccnumax.h"
#include "ccnumin.h"
#include "ccnulst.h"
#include "ccnuitr.h"
#include "ccnustr.h"
#include <string.h>
#include <stdio.h>

void output_list(String& str) {
   printf("%s\n",str.string());
}
void output_list(int var) {
   printf("%d ", var);
}

int input_list(String& str) {
   char tmpStr[80];
   if (scanf("%79s",tmpStr) == 1) {
      str.replace(tmpStr);
      return(1);
   }
   else
      return(0);
}

int input_list(int& var) {
  if(scanf("%d",&var) == 1 ) {
     return(1);
  }
  else
     return(0);
}

template <class Item> class IOList {
  public:
    IOList() : list() {}
    void write();
    void read(const char *msg);
    void append(Item item) {
      list.append(item);
    }
  private:
    List<Item> list;
};

template <class Item> void IOList<Item>::write() {
  Iterator<Item> iter(list);
  while (!iter.eol()) {
    output_list(++iter);
  }
  printf("\n");
}
```

Figure 14. C++ Template Program (Part 1 of 2)

```
template <class Item> void IOList<Item>::read(const char *msg) {
  Item item;
  printf("%s\n", msg);
  while (input_list(item)) {
  list.append(item);
  }
}

int main() {
  IOList<String> stringList;
  IOList<int>    intList;

  char line1[]= "This program will read in a list of";
  char line2[]= "strings, integers and real numbers";
  char line3[]= "and then print them out";

  stringList.append(line1);
  stringList.append(line2);
  stringList.append(line3);
  stringList.write();
  intList.read("Enter some integers (/* to terminate)");
  intList.write();

  String name1 = "Bloe, Joe";
  String name2 = "Jackson, Joseph";

  output_list(min(name1,name2));
  printf(" comes before ");
  output_list(max(name1,name2));
  printf("\n");


  int num1 = 23;
  int num2 = 28;

  output_list(min(num1,num2));
  printf(" comes before ");
  output_list(max(num1,num2));
  printf("\n");

  return(0);
}
```

Figure 14. C++ Template Program (Part 2 of 2)

Compiling, Binding, and Running the C++ Template Example

This section describes the commands to compile, bind and run the template example under CMS and the z/VM OpenExtensions shell.

Under CMS

Using the TEMPINC option: When using the TEMPINC option, which is the default, the template declaration and template definition files must be in the BFS. The template declaration files are:

- CCNULST.H
- CCNUITR.H
- CCNUMAX.H
- CCNUMIN.H

and should be put into the BFS with lowercase names, for example `ccnulst.h`. The template declaration files are:

- CCNULST.C
- CCNUITR.C
- CCNUMAX.C
- CCNUMIN.C

and should also be put into the BFS with lowercase names, for example `ccnulst.c`.

If the sample C++ template program (CCNUTMP.CPP) was stored in CCNUTMP CPP L, and the sample include file (CCNUSTR.H) was stored in CCNUSTR H L, the following set of commands would compile, bind, and run the source code, using the Language Environment:

```
GLOBAL LOADLIB SCEERUN                          1
CC CCNUTMP CPP L (LSEARCH(./)                   2
-- or, for XPLINK --
CC CCNUTMP CPP L (LSEARCH(./) XPLINK            2
CC ./tempinc/ccnulst.C (CXX                     3
CC ./tempinc/ccnuitr.C (CXX                     3
CC ./tempinc/ccnumax.C (CXX                     3
CC ./tempinc/ccnumin.C (CXX                     3
-- or, for XPLINK --
CC ./tempinc/ccnulst.C (CXX XPLINK              3
CC ./tempinc/ccnuitr.C (CXX XPLINK              3
CC ./tempinc/ccnumax.C (CXX XPLINK              3
CC ./tempinc/ccnumin.C (CXX XPLINK              3
cxx -o //ccnutmp.module //ccnutmp.text ./ccnulst.o
       ./ccnuitr.o ./ccnumax.o ./ccnumin.o                                  4
-- or, for XPLINK --
cxx -Wb,xplink -o //ccnutmp.module //ccnutmp.text
                  ./ccnulst.o ./ccnuitr.o ./ccnumax.o ./ccnumin.o   4
GLOBAL LOADLIB SCEERUN                          5
CCNUTMP                                         6
```

Figure 15. Commands to Compile, Bind, and Run a C++ Template Program under CMS (TEMPINC)

1 Makes the run-time library available to the compiler.

2 Compiles CCNUTMP CPP L and stores the object module in CCNUTMP TEXT A. Creates template instantiation files in the `./tempinc` directory.

3 Compiles the template instantiation files which will have been created in the `./tempinc` directory.

4 Using CCNUTMP TEXT A, and the template instantiation object modules in the BFS, generates an executable module called CCNUTMP MODULE using default options.

5 Makes the run-time library available to the exectuable module.

6 Runs CCNUTMP MODULE A using default options.

Using the TEMPLATEREGISTRY option: When using the TEMPLATEREGISTRY option, there is no restriction on where the template declaration and template definition files must be.

If the sample C++ template program (CCNUTMP.CPP) was stored in CCNUTMP CPP L, and the sample include file (CCNUSTR.H) was stored in CCNUSTR H L,

and the template declaration and template definition files were stored on the L disk or any other accessed CMS minidisk, the following set of commands would compile, bind, and run the source code, using the Language Environment:

```
GLOBAL LOADLIB SCEERUN            1
CC CCNUTMP CPP L (TEMPL           2
-- or, for XPLINK --
CC CCNUTMP CPP L (TEMPL XPLINK    2
CMOD CCNUTMP (C++                 3
-- or, for XPLINK --
CMOD CCNUTMP (XPLINK C++          3
GLOBAL LOADLIB SCEERUN              4
CCNUTMP                           5
```

Figure 16. Commands to Compile, Bind, and Run a C++ Template Program under CMS (TEMPLATEREGISTRY)

1 Makes the run-time library available to the compiler.

2 Compiles CCNUTMP CPP L and stores the object module in CCNUTMP TEXT A.

3 Using CCNUTMP TEXT A, created by the CC EXEC, generates an executable module called CCNUTMP MODULE using default options.

4 Makes the run-time library available to the exectuable module.

5 Runs CCNUTMP MODULE A using default options.

Under the z/VM OpenExtensions Shell

Using the TEMPINC option: If the sample C++ template program (CCNUTMP.CPP) was stored in ./ccnutmp.cpp, and the sample include file (CCNUSTR.H) was stored in ccnustr.h, and the template declaration and template definition files were stored in the root directory of the BFS, the following set of commands would compile, bind, and run the source code, using the Language Environment:

```
cxx -o ccnutmp ccnutmp.cpp                          1
-- or, for XPLINK --
cxx -Wc,xplink -Wb,xplink -o ccnutmp ccnutmp.cpp    1
cms global loadlib sceerun                            2
 ./ccnutmp                                            3
```

Figure 17. Commands to Compile, Bind, and Run a C++ Template Program under z/VM OpenExtensions (TEMPINC)

1 Compiles ccnutmp.cpp, compiles the template instantiation files which will have been created in the `./tempinc` directory, binds the created object modules, and stores the load module in ccnutmp.

2 Makes the run-time library available to the exectuable module.

3 Runs ccnutmp using default options.

Using the TEMPLATEREGISTRY option: If the sample C++ template program (CCNUTMP.CPP) was stored in ./ccnutmp.cpp, and the sample include file (CCNUSTR.H) was stored in ccnustr.h, and the template declaration and template definition files were stored in the root directory of the BFS, the following set of

commands would compile, bind, and run the source code, using the Language Environment:

```
cxx -Wc,templ -o ccnutmp ccnutmp.cpp                          1
-- or, for XPLINK --
cxx -Wc,templ,xplink -Wb,xplink -o ccnutmp ccnutmp.cpp        1
cms global loadlib sceerun                                      2
 ./ccnutmp                                                      3
```

Figure 18. Commands to Compile, Bind, and Run a C++ Template Program under z/VM OpenExtensions (TEMPLATEREGISTRY)

1 Compiles ccnutmp.cpp, binds the created object module, and stores the load module in ccnutmp.

2 Makes the run-time library available to the exectuable module.

3 Runs ccnutmp using default options.

Compiler Options

This chapter describes the options that you can use to alter the compilation of your program. For information on compiler options when compiling under OpenExtensions, refer to “Compiler Options under OpenExtensions” on page 49.

Specifying Compiler Options

You can override your installation default options when you compile your C or C++ program, by specifying an option in one of the following ways:

- In the option list when you invoke the IBM-supplied CC EXEC. See “Syntax of the CC EXEC” on page 56 for details.
- In an options file. See “OPTFILE | NOOPTFILE” on page 45 for details.
- In a `#pragma` options preprocessor directive within your source file. See “Specifying Compiler Options Using #pragma options” on page 39 for details.

  Compiler options specified on the command line can override compiler options used in `#pragma` options.

If two contradictory options are specified, the last one specified is accepted and the first ignored.

If you use one of the following compiler options, the option name is inserted at the bottom of your object module indicating that it was used:

AGGRCOPY	
ALIAS	(C compile only)
ANSALIAS	
ARCHITECTURE	
ARGPARSE	
BITFIELD	
CHARS	
COMPACT	
COMPRESS	
CONVLIT	

CSECT	
CVFT	(C++ compile only)
DLL	
EXECOPS	
EXPORTALL	
FLOAT	
GOFF	
GONUMBER	
IGNERRNO	
INITAUTO	
INLINE	
KEYWORD	(C++ compile only)
LANGLVL	
LIBANSI	
LOCALE	
LONGNAME	
MAXMEM	
OBJECTMODEL	
OPTIMIZE	
PLIST	
REDIR	
RENT	(C compile only)
ROCONST	
ROSTRING	
ROUND	
RTTI	(C++ compile only)
SERVICE	
SPILL	
START	
STRICT	
STRICT_INDUCTION	
TARGET	
TEMPLATERECOMPILE	(C++ compile only)
TEMPLATEREGISTRY	(C++ compile only)
TMPLPARSE	(C++ compile only)
TEST	
TUNE	
UPCONV	(C compile only)
XPLINK	

Specifying Compiler Options Using #pragma options

You can use the #pragma options preprocessor directive to override the default values for compiler options. Remember that compiler options specified on the command line can override compiler options used in #pragma options. For complete details on the #pragma options preprocessor directive, see the *C/C++ Language Reference*.

The #pragma options preprocessor directive must appear before the first C statement in your input source file. Only comments and other preprocessor directives can precede the #pragma options directive, and only the options listed below can be specified. If you specify a compiler option that is not given in the following list, the compiler generates a warning message and the option is ignored.

```
AGGREGATE                   OBJECT
ALIAS                       OPTIMIZE
ANSALIAS                    RENT
ARCHITECTURE                SERVICE
CHECKOUT                    SPILL
ENUMSIZE                    START
GONUMBER                    TEST
IGNERRNO                    TUNE
INLINE                      UPCONV
LIBANSI                     XREF
MAXMEM
```

Notes:

1. When you specify conflicting attributes either explicitly or implicitly by the specification of other options, the last explicit option is accepted. No diagnostic message is issued to indicate that any options are overridden.
2. When you specify the SOURCE compiler option on the command line, your listing will contain an options list indicating the options in effect at invocation. The values in the list are the options specified on the command line or the default options specified at installation. These values do not reflect any options that are specified in the #pragma options directive.

Compiler Option Defaults

You can alter the compilation of your program by specifying compiler options when you invoke the compiler or when you use the preprocessor directive, #pragma options, in your source program. Options that you specify when you invoke the compiler override installation defaults or compiler options specified through a #pragma options directive.

The defaults of the compiler options supplied by IBM can be changed to other selected defaults when C/C++ for z/VM is installed. To determine the current defaults, compile a program with only the SOURCE compiler option specified. In the listing generated, you can view the options that are in effect at invocation; that is, the settings that result from the interaction of the command-line options and the defaults that were specified at installation. The listing does not reflect options specified in #pragma options in the source file being compiled.

Summary of Compiler Options

Please refer to the corresponding section in the *z/OS C/C++ User's Guide*.

Description of Compiler Options

Refer to the corresponding section in the *z/OS C/C++ User's Guide* for details of specific compiler options.

For the most part, the compiler options will work as described in the *z/OS C/C++ User's Guide*. Refer to that manual for information on specific options. However, there are differences and restrictions in the operation of some of the options for CMS. The following sections detail these.

z/OS C/C++ Options Not Supported in C/C++ for z/VM.

The following z/OS C/C++ compiler options are not supported in C/C++ for z/VM:

IPA Interprocedural Analysis

ASCII ASCII support

z/OS C/C++ Options With Restricted Operation In C/C++ for z/VM.

TEMPINC: Under z/OS, this option specifies, either specifically or by default, a location where the compiler will place all C++ template instantiation files. If the source file resides in a dataset, the default location is a PDS with a low-level qualifier of TEMPINC. If the source resides in the HFS, the default location is the HFS directory `./tempinc`. A PDS name or HFS directory name can be specified to override this.

Under CMS, because of the way the compiler is implemented, the source file always appears to the compiler as residing in the BFS. Therefore, the default TEMPINC location is always the BFS. Also, this option will only work if all of the template declaration files (.h) and template definition files (.c) are in the BFS. The template instantiation files will then be written to the `./tempinc` directory by default or a directory specified on the TEMPINC option. A BFS directory is the only valid location on the TEMPINC option under CMS. That is, the location name must contain a '/', for example:

```
TEMPINC(./fred)
```

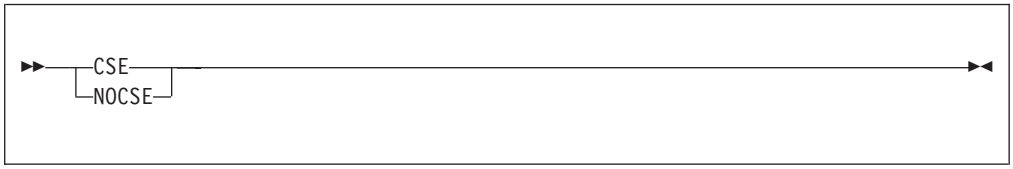
or

```
TEMPINC(fred/jim)
```

If the location is not a BFS directory, the compiler will issue a `CCN6001` message.

z/OS C/C++ Options With Operational Differences In C/C++ for z/VM.

CSECT | NOCSECT:



This option does not accept a qualifier. If a qualifier is specified it is ignored.

If `CSECT` is specified, it will name the code, static and test sections of the object module as *basename#suffix*, where:

basename Is one of the following:

- The file name of the primary source file, if it is a CMS record file
- The source file name, with path and extension information removed, if it is a BFS file

suffix Is one of the following:

C For code CSECT

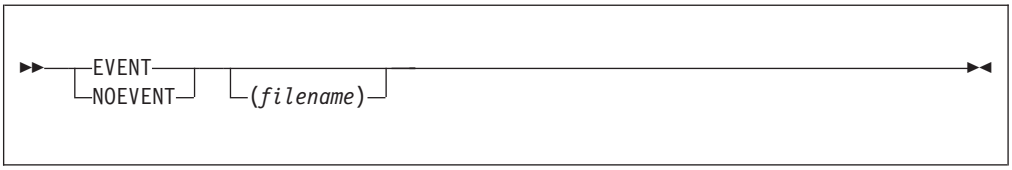
S For static CSECT

T For test CSECT

Specifying CSECT allows the compiler to generate long CSECT names. For NOGOFF, if the compiler option LONGNAME is not in effect when CSECT is specified, the compiler turns it on, and issues an informational message. For GOFF, both NOLONGNAME and LONGNAME options are supported.

When CSECT is specified, the code, data and test CSECTs are always generated. The test CSECT has content only when the TEST option is also specified.

EVENTS | NOEVENTS:



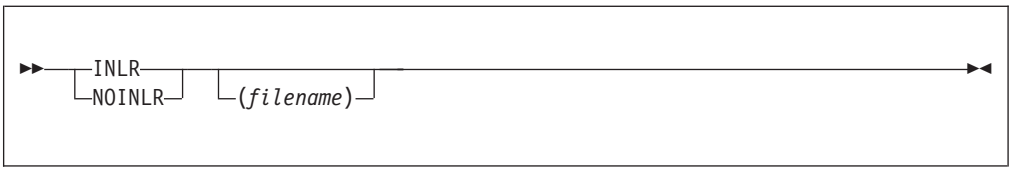
The EVENTS option creates an events file that contains error information and source file statistics.

EVENTS(filename) places the events information in the specified file. filename can be a CMS record or BFS file. If you do not specify a file name for the EVENTS option, the compiler generates a file name as follows:

- For CMS source files, the events information is written to a file that has the name of the source file and a file type of SYSEVENT.
- For BFS source files, the events information is written to a file that has the name of the source file and a .err extension.

The compiler ignores #line directives when the EVENTS option is active, and issues a warning message.

INLRPT | NOINLRPT:



If you use the OPTIMIZE option, you can also use INLRPT to specify that the compiler generate a report as part of the compiler listing. This report provides the status of subprograms that were inlined, specifies whether they were inlined or not and displays the reasons for the action of the compiler.

You can specify filename for the inline report output file. filename can be a CMS record or BFS file. If you do not specify a file name for the INLR option, the compiler generates a file name as follows:

- For CMS record source files, the report is created in a file that has the source file name, file type LISTING, and file mode A.

- For BFS source files, the report is created in a BFS file that has the source file name with a `.lst` extension.

The `NOINLR` option can optionally take a filename suboption. This filename then becomes the default. If you subsequently use the `INLR` option without filename, the compiler uses the filename that you specified in the earlier specification or `NOINLR`. For example,

```
CC HELLO (NOINLR(/hello.lis) INLR OPT
```

is the same as specifying:

```
CC HELLO (INLR(/hello.lis) OPT
```

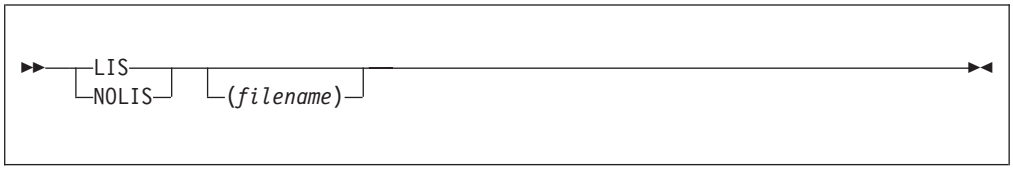
If you specify this multiple times, the compiler uses the last specified option with the last specified suboption. The following two specifications have the same result:

```
CC HELLO (NOINLR(/hello.lis) INLR(/n1.lis)  NOINLR(/test.lis) INLR

CC HELLO (INLR(/test.lis)
```

If you specify filenames with the `SOURCE`, `LIST` or `INLRPT` options, all the listing sections are combined into the last file name specified.

LIST / NOLIST:



The `LIST` option instructs the compiler to generate a listing of the machine instructions in the object module (in a format similar to assembler language instructions) in the compiler listing.

`LIST(filename)` places the compiler listing in the specified file. `filename` can be a CMS record or BFS file. If you do not specify a file name for the `LIST` option, the compiler generates a file name as follows:

- For CMS record source files, the listing is created in a file that has the source file name, file type LISTING, and file mode A.
- For BFS source files, the listing is created in a BFS file that has the source file name with a `.lst` extension.

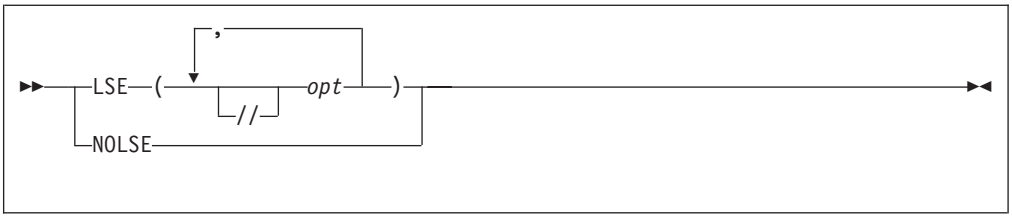
The `NOLIST` option optionally takes a filename suboption. This filename then becomes the default. If you subsequently use the `LIST` option without a filename suboption, the compiler uses the filename that you specified in the earlier `NOLIST`. For example, the following specifications have the same effect:

```
CC HELLO (NOLIST(/hello.lis) LIST

CC HELLO (LIST(/hello.lis)
```

If you specify filenames with the `SOURCE`, `LIST` or `INLRPT` options, all the listing sections are combined into the last file name specified.

LSEARCH / NOLSEARCH:

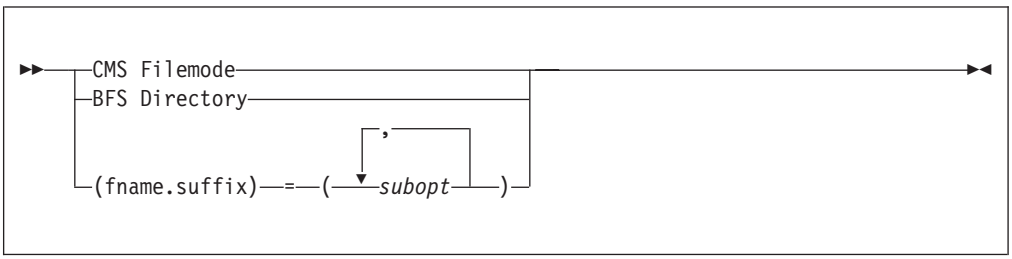


The LSEARCH option directs the preprocessor to look for user include files in the specified libraries in the VM/CMS MACLIBs, on the specified minidisks, or in the specified BFS directories. User include files are files associated with the #include "filename" format of the #include preprocessor directive. See “Using Include Files” on page 62 for a description of the #include preprocessor directive.

For further information on library search sequences, see “Search Sequences for Include Files” on page 67.

You must use the double slashes (//) to specify non-BFS searches when the OE compiler option is specified. (You may use them regardless of the OE option.)

Parts of the #include filename are appended to each LSEARCH opt to search for the include file. opt has the format:



opt specifies one of the following:

- A file mode where the sequential disk search for the user include file begins
- A BFS path name indicating the directory that should be searched for the include file
- A specification of the form (fname.suffix) = (subopt,subopt,...) where:
  - fname is the name of the include file or *
  - suffix is the suffix of the include file or *
  - subopt indicates a subpath to be used in the search for the include files that match the pattern of fname.suffix and should appear at least once. The possible values are:
    - LIB([mac,...]) where each mac is a MACLIB name that should be searched in the same order as in the list. The format of the name is either that of a ddname (form DD:name) or fn.ft.fm, where the ft must be MACLIB and the default fm is *.
    - NOLIB specifies that all LIB(...) previously specified for this pattern should be ignored at this point. For example, (*.h)=(LIB(n1.MACLIB),NOLIB,LIB(n4.MACLIB)) is equivalent to (*.h)=(LIB(n4.MACLIB)).

    When the #include filename matches the pattern of fname.suffix, the search continues according to the subopts in the order given. An asterisk (*) in fname

or suffix matches anything. If the file is not found, other searches are attempted according to the remaining options in LSEARCH.

The MACLIBs are searched in the same order for the include file with * (asterisk) matching anything.

If a file mode is also specified using the SEARCH option, the disks specified by the LSEARCH option are searched first. If the user include file is not located on any of the LSEARCH disks, the disks in the SEARCH option are scanned, in the standard CMS search order, for the user include file.

If no disk is specified, the file mode A will be added to the end of the LSEARCH options.

For more information on the search paths, see "Search Sequences for Include Files" on page 67.

Under CMS, the NOLSEARCH option instructs the preprocessor to perform the standard CMS search for user include files.

Note: If the filename in the #include directive is in absolute form, searching is not performed. See "Determining If filename Is In Absolute Form" on page 64 for more details on absolute #include filename.

Specifying Byte File System (BFS) Files: When specifying BFS library searches, do not put double slashes at the beginning of the LSEARCH opt. Use pathnames separated by slashes (/) in the LSEARCH opt for a BFS library. When the LSEARCH opt does not start with double slashes, any single slash in the name indicates a BFS library. If you do not have path separators (/), then setting the OE compiler option on indicates that this is a BFS library; otherwise the library is interpreted to be a CMS library.

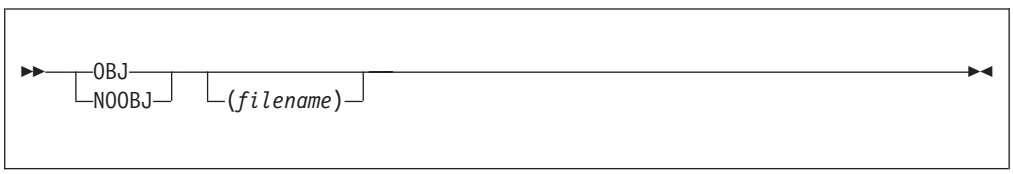
The opt specified for LSEARCH is combined with the filename in #include to form the include file name, for example:

```
LSEARCH(/u/mike/myfiles)
#include "new/headers.h"
```

Resulting BFS file name is:

```
/u/mike/myfiles/new/headers.h
```

OBJECT | NOOBJECT:



The OBJECT option specifies whether the compiler is to produce an object module.

The GOFF compiler option specifies the object format that will be used to encode the object information.

You can specify OBJECT(filename) to place the object module in that file. filename can be:

- a CMS record file
- a single-letter mode to which the object module is stored as `filename TEXT fm`
- a BFS file
- a fully qualified path name
- a path name relative to the current working directory

If you do not specify a file name for the `OBJECT` option, the compiler generates a file name as follows:

- For CMS record source files, the listing is created in a file that has the source file name, file type TEXT, and file mode A.
- For BFS source files, the listing is created in a BFS file that has the source file name with a `.o` extension.

The `NOOBJ` option can optionally take a filename suboption. This filename then becomes the default. If you subsequently use the OBJ option without a filename suboption, the compiler uses the filename that you specified in the earlier `NOOBJ`. For example, the following specifications have the same result:

```
CC HELLO (NOOBJ(/hello.obj) OBJ

CC HELLO (OBJ(/hello.obj)
```

If you specify `OBJ` and `NOOBJ` multiple times, the compiler uses the last specified option with the last specified suboption. For example, the following specifications have the same result:

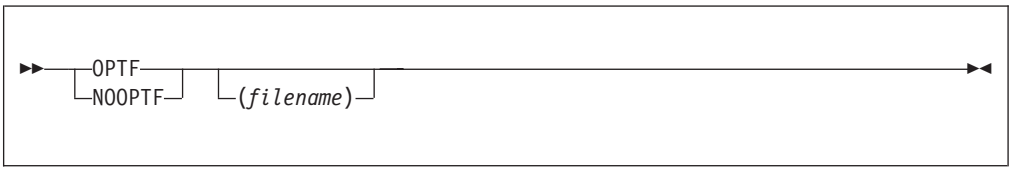
```
CC HELLO (NOOBJ(/hello.obj) OBJ(/n1.obj)  NOOBJ(/test.obj) OBJ

CC HELLO (OBJ(/test.obj)
```

If you request a listing by using the `SOURCE`, `INLRPT`, or `LIST` option, and you also specify `OBJECT`, the name of the object module is printed in the listing prolog.

You can specify this option using the `pragma` option directive for C.

OPTFILE | NOOPTFILE:



The `OPTFILE` option directs the compiler to look for compiler options in the file specified. `OPTF(filename)` specifies the name of the options file where your compiler options are defined. `filename` can be a CMS record or BFS file. The compiler opens `filename` as it is specified. If `filename` is an invalid file name, or if the file does not exist, the compiler does not issue an error message. For example, specifying:

```
CC cpgma (OPTFILE(myopts)
```

does not cause an error, but file `myopts` is not opened.

Specifying:

```
CC cpgma (OPTFILE(myopts optfile)
```

opens options file `myopts optfile`. Under the OpenExtensions shell, `filename` is a BFS file. If `filename` is not specified, DD:SYSOPTF is used.

The `NOOPTF` option can optionally take a `filename` suboption. This filename then becomes the default. If `NOOPTF(filename)` is specified and a subsequent `OPTF` option is used without a `filename` suboption, the filename specified in the previous `NOOPTF` is used. For example,

```
CC HELLO (NOOPTF(hello.opt) OPTF
```

is equivalent to specifying:

```
CC HELLO (OPTF(hello.opt)
```

The options are specified in a free format with the same syntax as they would have on the command line. Everything specified in the file is taken to be part of a compiler option (except for the continuation character) and unrecognized entries are flagged. Nothing on a line is ignored.

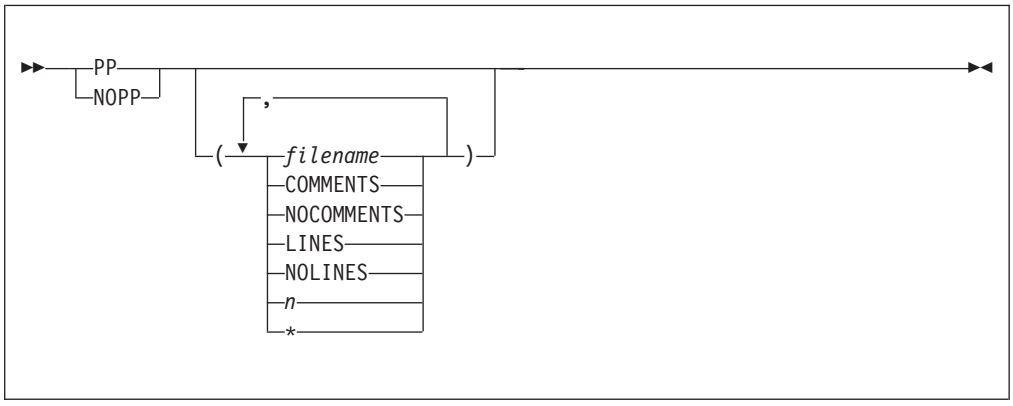
If the record format of the options file is fixed and the record length is greater than 72, columns 73 to the end-of-line are treated as sequence numbers and are ignored.

Notes:

1. You cannot nest the `OPTFILE` option. If the `OPTFILE` option is also used in the file specified by another `OPTFILE` option, it is ignored.
2. If `NOOPTFILE` is specified after a valid `OPTFILE`, it does not undo the effect of the previous `OPTFILE`.
3. If the file cannot be opened or cannot be read, **NO** warning message will be issued and the `OPTFILE` option will be ignored.
4. The options file can be an empty file.

The `OPTFILE` option is added to the options section of the compiler-generated listing file.

PPONLY | NOPPONLY:



The `PPONLY` option specifies that only the preprocessor is to be run against the source file. This output of the preprocessor consists of the original source file with all the macros expanded and all the include files inserted. It is in a format that can be compiled.

The suboptions are:

COMMENTS|NOCOMMENTS

The COMMENTS suboption preserves comments in the preprocessed output. The default is NOCOMMENTS.

LINES | NOLINES

The LINES suboption issues `#line` directives at include file boundaries, block boundaries and where there are more than 3 blank lines. The default is NOLINES.

filename The name for the preprocessed output file. `filename` can be a CMS record or BFS file. If a `filename` is not specified for the `PPONLY` option, the compiler writes the preprocessed output as follows:

- For CMS record source files, the preprocessed output is written to a file that has the source file name and file type EXPAND.
- For BFS source files, the preprocessed output is written to a BFS file that has the source file name with an `.i` extension.

n If a parameter n, which is an integer between 2 and 32760 inclusive, is specified, all lines are folded at column n.

***** If an asterisk (*) is specified, the lines are folded at the maximum record length of 32760. Otherwise, all lines are folded to fit in the output file, based on the record length of the output file.

The PPONLY suboptions are cumulative. If you specify suboptions in multiple instances of PPONLY and NOPPONLY, all the suboptions are combined and used for the last occurrence of the option. For example, the following three specifications have the same result:

```
CC HELLO (NOPPONLY(/aa.exp) PPONLY(LINES) PPONLY(NOLINES)

CC HELLO (PPONLY(/aa.exp,LINES,NOLINES)

CC HELLO (PPONLY(/aa.exp,NOLINES)
```

All `#line` and `#pragma` preprocessor directives (except for margins and sequence directives) remain. When you specify `PPONLY(*)`, `#line` directives are generated to keep the line numbers generated for the output file from the preprocessor similar to the line numbers generated for the source file. All consecutive blank lines are suppressed.

If you specify the `PPONLY` option, the compiler turns on the `TERMINAL` option. If you specify the `SHOWINC`, `XREF`, `AGGREGATE`, or `EXPMAC` options with the `PPONLY` option, the compiler issues a warning, and ignores the options.

If you specify the `PPONLY` and `LOCALE` options, all the `#pragma filetag` directives in the source file are suppressed. The compiler generates its `#pragma filetag` directive at the first line in the preprocessed output file in the following format:

```
??=pragma filetag ("locale code page")
```

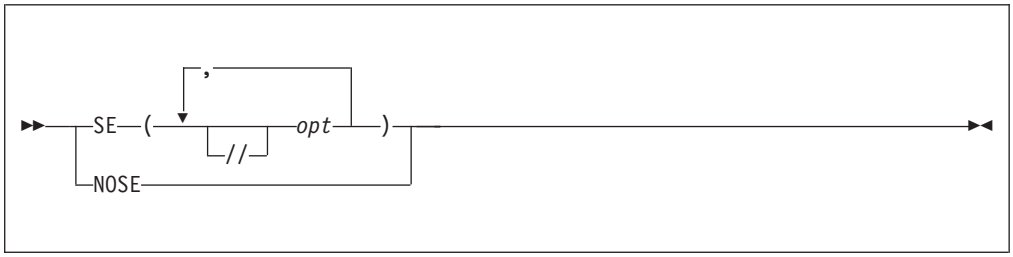
In the above, ??= is a trigraph representation of the # character.

The code page in the pragma is the code set that is specified in the `LOCALE` option. For more information on locales, refer to the *z/OS C/C++ Programming Guide*.

The `NOPPONLY` option specifies that both the preprocessor and the compiler are to be run against the source file.

If you specify both `PPONLY` and `NOPPONLY`, the last one that is specified is used.

SEARCH | NOSEARCH:



The SEARCH option directs the preprocessor to look for system include files in the specified libraries in the VM/CMS MACLIBs, on the specified minidisks, or in the specified BFS directories. System include files are those files associated with the #include <filename> format of the #include preprocessor directive. See "Using Include Files" on page 62 for a description of the #include preprocessor directive.

For further information on library search sequences, see "Search Sequences for Include Files" on page 67.

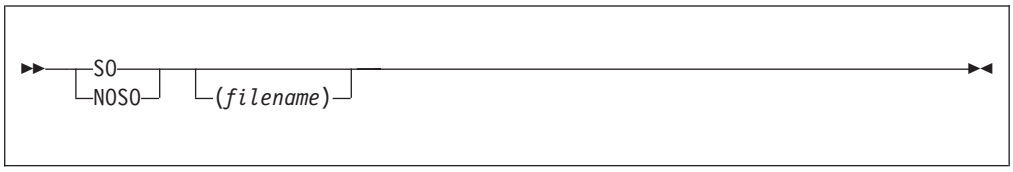
The suboptions for the SEARCH option are identical to those for the LSEARCH option, as described in "LSEARCH | NOLSEARCH" on page 42.

Any NOSEARCH option cancels all previous SEARCH specifications and any SEARCH options following it will be used. When several SEARCH compiler options are specified, all the libraries in these SEARCH options are used to find the user include files.

The NOSEARCH option instructs the preprocessor to perform the standard CMS search for system include files.

Note: If the filename in the #include directive is in absolute form, searching is not performed. See "Determining If filename Is In Absolute Form" on page 64 for more details on absolute #include filename.

SOURCE | NOSOURCE:



The SOURCE option generates a listing that shows the original source input statements plus any diagnostic messages.

If you specify SOURCE(filename), the compiler places the listing in the file that you specified. filename can be a CMS record or BFS file. If you do not specify a file name for the SOURCE option, the compiler constructs the file name as follows:

- For CMS record source files, the listing is created in a file that has the source file name, file type LISTING, and file mode A.
- For BFS source files, the listing is created in a BFS file that has the source file name with a .lst extension.

The NOSOURCE option can optionally take a filename suboption. This filename then becomes the default. If you subsequently use the SOURCE option without a filename suboption, the compiler uses the filename that you specified in the earlier NOSOURCE. For example, the following specifications have the same result:

```
CC HELLO (NOSO(/hello.lis) SO

CC HELLO (SO(/hello.lis)
```

If you specify SOURCE and NOSOURCE multiple times, the compiler uses the last specified option with the last specified suboption. For example, the following specifications have the same result:

```
CC HELLO (NOSO(/hello.lis) SO(/n1.lis)  NOSO(/test.lis) SO

CC HELLO (SO(/test.lis)
```

If you specify filenames with the SOURCE, LIST or INLRPT options, all the listing sections are combined into the last file name specified.

Using the C Compiler Listing

Please refer to the corresponding section in the *z/OS C/C++ User's Guide*.

Using the C++ Compiler Listing

Please refer to the corresponding section in the *z/OS C/C++ User's Guide*.

Compiler Options under OpenExtensions

This chapter discusses the compiler options that you can use when compiling under z/VM OpenExtensions. For information about compiler options under VM/CMS, refer to "Compiler Options" on page 37.

Specifying Compiler Options Using c89/cxx

The c89 and cxx utilities are the z/VM OpenExtensions interface to the C/C++ for z/VM compiler. When you issue c89 for a C application program, or cxx for a C++ application program, the utility passes information about the application program and the compiler options to the C/C++ for z/VM compiler for processing.

The c89 and cxx utilities select specific values for most compiler options. You can cause them to change the settings of those C/C++ compiler options that they have corresponding flags (options) for. If you want to pass other C/C++ compiler options to the C/C++ for z/VM compiler, use the -W option. If you used the options -E, -g, -s, or -O you cannot override the compiler options forced by the c89 or cxx utility. This holds true even when using the -W option to explicitly pass C/C++ for z/VM compiler options.

C/C++ for z/VM compiler options are summarized in "Compiler Option Defaults" on page 39 and described in detail in "Description of Compiler Options" on page 39. For more information on z/VM OpenExtensions commands, refer to the *z/VM OpenExtensions Command Reference*.

c89/cxx Default Compiler Settings

c89 overrides the default settings for the C/C++ for z/VM compiler options. The overridden defaults are:

- DEFINE(errno=(*__errno()))
- DEFINE(_POSIX_SOURCE=1)

- DEFINE(_POSIX1_SOURCE=2)
- DEFINE(_POSIX_C_SOURCE=2)
- LANGLVL(ANSI)
- OE
- RENT

c89 Selectable Compiler Settings

Format

```
c89 [-cgsEOV]
      [-D name[=value]]  .... [-U name]...
      [-W c,opt[,opt]...  ]...
      [-o outfile]
      [-I directory]... [-L   directory]...
      [file.c]... [file.a]... [file.o]...
      [-l libname]...
cxx [-+cgsEOV]
      [-D name[=value]]  .... [-U name]...
      [-W c,opt[,opt]...  ]...
      [-o outfile]
      [-I directory]... [-L   directory]...
      [file.c]... [file.a]... [file.o]...
      [-l libname]...
```

Description

c89/cxx Option	Compiler Option
-+ (cxx only)	All source files are to be recognized as C++ source files.
-c	Compilation only
-D	Define preprocessor macros.
-E	Run the C preprocessor only (do not generate an object file or run the linkage editor) and copy output source to stdout.
-g	Generate symbolic information with the compiled object. The `c89 -s` option, the default, indicates that no debugging information or line number tables be generated.
-I	Specify where to search for C include files. The search path is supplied as a value on the option. For example: `-I /usr/hankvp/bin/hdrs`
-L	Specify where to search for archive files specified by the `-I` option.
-O	Set an optimization level and place functions at their point of call.
o	Write the executable file to `outfile`.
-s	Do not generate symbolic information with the compiled object.
-U	Undefine preprocessor macros (including `c89` default macro definitions).
-V	Write a ″verbose″ listing to stdout. Listings are generated by the compiler and binder. The information in the compiler listing corresponds to those compiler options set by the `c89 -V option`. For a complete description of the effect of each compiler option, see “Description of Compiler Options” on page 39.

-W Pass compiler or module build options. Phase `0` or `c` specifies the compile phase, and phase `b` specifies the module build phase. The module build phase is binder processing to create the module file. To pass options to the BIND command, the module build option must be `b`. For example, to pass the `LANGLVL` option to the compiler, specify:

```
c89 -W 0,langlvl(extended)
```

and to write the binder map to stdout, specify:

```
c89 -W b,b,map file.c
```

For a detailed description of the c89 options, refer to the *z/VM OpenExtensions Command Reference*.

`c89` uses the following compiler option settings if the `c89` option listed is specified (more than one compiler option may be specified by a particular c89 option):

Table 1. c89 Option and Corresponding Compiler Option

c89 Option	**Compiler Options(s)**
`D` *value*	DEFINE(*value*)
Note: The `c89 -U` *value* option causes `c89` to not specify a corresponding `DEFINE`(*value*) compiler option.	
`-E`	`PPONLY(1024)`
`-g`	`TEST(ALL)` `GONUMBER`
`-I`*value*	`SEARCH`(*value*)

`-O`	`INLINE(NOAUTO,NOREPORT,250,1000)`
	`NOMEMORY`
	`OPTIMIZE(2)`

`-s`	`NOGONUMBER` `NOTEST`

`-V`	`AGGREGATE`
	`CHECKOUT(ALL,NOEXTERN,NOPPCHECK,NOPPTRACE)`
	`FLAG(I)`
	`LIST`
	`OFFSET`
	`SHOWINC`
	`SOURCE`
	`XREF`

Note: Use of the `c89 -V` option *may* result in a return code of 4 from the compile step when the return code should be 0. This is because of the specification of the `CHECKOUT` option. Also, the specification of `FLAG(I)` may cause additional informational messages to be directed to `stderr`.

Feature Test Macros

For information on how to use the feature test macros, refer to the *C/C++ for z/VM Run-Time Library Reference*.

Run-Time Options

This chapter describes Run-Time options and the #pragma runopts preprocessor directives available to you with C/C++ for z/VM and Language Environment. For information on Run-Time options under OpenExtensions, refer to “Specifying Run-Time Options under OpenExtensions” on page 85.

Specifying Run-Time Options

To allow your application to recognize run-time options, either the EXECOPS compiler option, or the #pragma runopts(execops) directive must be in effect. The default compiler option is EXECOPS.

You can specify run-time options as follows:

- On the command line when you invoke your program under VM/CMS
- At compile time, on a #pragma runopts directive in your main program

If EXECOPS is in effect, use a slash '/' to separate run-time options from arguments that you pass to the application. For example:

```
PGMX STORAGE(FE,FE,FE)/PARM1 PARM2 PARM3
```

If EXECOPS is in effect, Language Environment interprets the character string that precedes the slash as run-time options. It passes the character string that follows the slash to your application as arguments. If no slash separates the arguments, Language Environment interprets the entire string as an argument.

If EXECOPS is not in effect, Language Environment passes the entire string to your application.

If you specify two or more contradictory options (for example in a #pragma runopts statement), the last option that is encountered is accepted. Run-time options that you specify at execution time have higher precedence than those specified at compile time.

For more information on the precedence and specification of run-time options for applications that are compiled with the Language Environment, refer to the *Language Environment Programming Reference*.

Run-Time Options Using the Language Environment

You can use the #pragma runopts preprocessor directive to specify Language Environment run-time options, including ARGPARSE, ENV, PLIST, REDIR, and EXECOPS, which have matching compiler options. If you specify the compiler option, it has precedence over the #pragma runopts directive.

When the run-time option EXECOPS is in effect, you can specify run-time options at execution time, as previously described. These options override run-time options that you compiled into the program by using the #pragma runopts directive.

The #pragma runopts directive can appear in any file: main, include, or source. You can specify multiple run-time options per directive or multiple directives per compilation unit. If you want to specify the ARGPARSE or REDIR options, the #pragma runopts directive must be in the same compilation unit as main().

When you specify multiple instances of #pragma runopts in separate compilation units, the compiler generates a CSECT for each compilation unit that contains a

`#pragma runopts` directive. When you bind multiple compilation units that specify `#pragma runopts`, the binder takes only the first CSECT, thereby ignoring your other option statements. Therefore, you should always specify your `#pragma runopts` directive in the same source file that contains the function `main()`.

For more information on the `#pragma runopts` preprocessor directive, see the *C/C++ Language Reference*.

Chapter 3. Compiling, Binding and Running

Compiling

This chapter describes how to compile your program using the C/C++ for z/VM compiler and the Language Environment under VM/CMS. For information on compiling your program under OpenExtensions, refer to “z/VM OpenExtensions: Compiling a C/C++ Program” on page 81.

The C/C++ for z/VM compiler analyzes the C/C++ source program and translates the source code into machine instructions known as *object code*.

To compile your C/C++ source program using the C/C++ for z/VM compiler, you must have access to the Language Environment because the compiler calls functions in the library to compile code.

Invoking the C/C++ for z/VM Compiler

When you invoke the C/C++ for z/VM compiler, the operating system automatically tries to locate and execute the compiler. The location of the compiler is determined by the system programmer who installed the product. The compiler may be in a

nucleus extension, in a *Named Saved Segment* (NSS), or on a minidisk. In either instance, you only need to ensure that you have access to the C compiler version that you want to use.

The C/C++ for z/VM compiler can be invoked under VM/CMS using the IBM supplied `CC` EXEC.

The C/C++ for z/VM compiler compiles source code using the Language Environment. You must ensure that the load libraries that contain C/C++ for z/VM, Language Environment, and VM/CMS library routines are available. The Run-Time libraries are needed for compilation, because the compiler calls functions from the libraries. The `GLOBAL` command is used to link to the libraries. The libraries may be in a nucleus extension, an NSS, or in the `GLOBAL LOADLIB` list. For more information on how to make libraries available for execution, refer to "Making the Run-Time Libraries Available for Execution" on page 78. The following examples assume that the default names (which can be changed by the system programmer during installation) are used.

GLOBAL Command for Using the Language Environment Library

The `GLOBAL` commands to make the library available to compile, bind, and run a program are as follows:

- To run the compiler:

  ```
  GLOBAL LOADLIB USERLIB SCEERUN
  ```

- To bind your C object code:

  ```
  GLOBAL TXTLIB USERLIB SCEELKED
  ```

- To bind your C++ object code:

  ```
  GLOBAL TXTLIB USERLIB SCEELKED SCEECPP
  ```

- To run your C or C++ module:

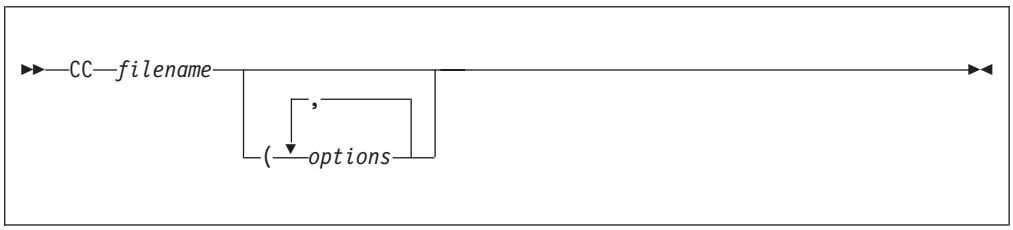
  ```
  GLOBAL LOADLIB USERLIB SCEERUN
  ```

where `USERLIB` represents any user load or text libraries.

Note: The SCEECPP text library is part of Language Environment and contains the base C++ linkedit routines.

Syntax of the CC EXEC

The syntax of the CC EXEC is:



where:

filename Specifies the name of the source file to be compiled. The source file can be a CMS record or BFS file.

options Specifies the compiler options to use during compilation. If no compiler options are specified, the default settings are used.

For a description of the compiler options that you can specify when invoking the CC EXEC, refer to “Description of Compiler Options” on page 39.

Specifying the Input File

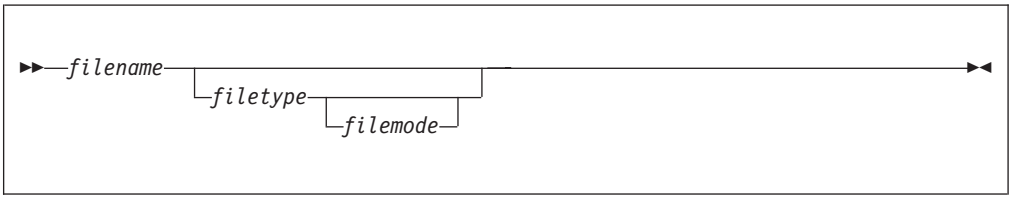
Input for the compiler consists of:

- Your C/C++ source program
- C/C++ for z/VM standard header files
- Your header files

The primary input to the compiler is the first argument passed to the `CC` exec. Your C/C++ source may be in a CMS record or BFS file. The secondary input to the compiler consists of files identified by `#include` preprocessor directives in the input. For more information on `#include` files, see “Using Include Files” on page 62.

The output that the compiler generates is based on the primary source file input.

CMS Record Files: To specify a CMS record file as your primary source file, use the following syntax:



You must always specify the source file name following the `CC` keyword. If the file type is not C, the file type must also be specified on the `CC` EXEC. If you do not specify the file mode, the currently accessed minidisks are searched in the standard VM/CMS search order. The file that is compiled is the first one encountered in the disk search. For example, if you have a file called `TWICE C` on both your `B` and `Y` minidisks, and the `Y` minidisk is not accessed as an extension of the `A` disk, `TWICE C B` is compiled if you do not specify the file mode. Note also that if you specify the file mode, you must also specify the file type.

```
KNOWN:   - The file name is SALARY.
         - The file type is C.
         - The file mode is A.

USE THE FOLLOWING COMMAND:
         CC SALARY
Result:  The object module generated will have file name
         SALARY, a file type of TEXT, and file mode A.
```

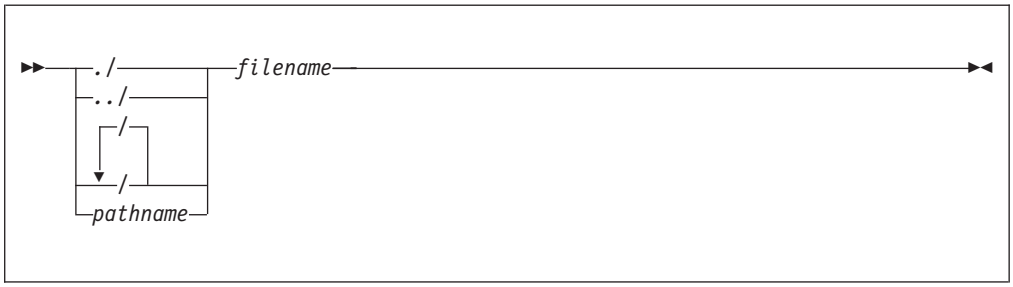
Figure 19. Specifying a CMS Record Input File under VM/CMS (Example 1)

```
KNOWN:   - The file name is INCOME
         - The file type is NET
         - The file mode is Y

USE THE FOLLOWING COMMAND:
         CC INCOME NET Y
Result:  The object module generated will have file name INCOME,
         a file type of NET, and file mode A.
```

Figure 20. Specifying a CMS Record Input File under VM/CMS (Example 2)

BFS Files: You can also use the `CC` EXEC to compile source that is in BFS files. To specify a BFS file as your primary source file, use the following syntax:



where:

./	Specifies the current directory.
../	Specifies the previous directory.
/	Specifies the beginning of an absolute path name.
pathname	Specifies all directories leading to the file.
filename	Specifies the name of the source file.

When you use the CC EXEC, you must use unambiguous BFS source file names. For example, the compiler treats the following input files as BFS files:

```
CC ./test/hello.c
CC /u/david/test/hello.c
CC test/hello.c
CC ///hello.c
CC ../test/hello.c
```

If *filename* is not in the *pathname* format with single slashes, the compiler treats the file as non-BFS input. The following input files are treated as non-BFS files:

```
CC hello.c
CC //hello.c
```

Refer to the *z/VM OpenExtensions User's Guide* for complete information on working with BFS files.

```
KNOWN:   - The current working directory is /u/proga.
         - The file name is myprog.c.

USE THE FOLLOWING COMMAND:
         CC ./myprog.c
Result:  The object module generated will be in the current
         working directory and have a file name
         myprog.o.
```

Figure 21. Specifying a BFS Input File under VM/CMS (Example 1)

```
KNOWN:   - The file name is myprog.c in directory
           /u/boris/progs.
         - The current working directory is /u/proga.

USE THE FOLLOWING COMMAND:
         CC /u/boris/progs/myprog.c
Result:  The object module generated will be myprog.o
         in the current working directory /u/proga.
```

Figure 22. Specifying a BFS Input File under VM/CMS (Example 2)

For information on compiling programs under z/VM OpenExtensions, see “z/VM OpenExtensions: Compiling a C/C++ Program” on page 81.

Specifying Compiler Options

There are many compiler options that you can specify when you compile using the CC EXEC. They are described in “Description of Compiler Options” on page 39.

The following examples show you how to override the default options when compiling under VM/CMS. When you specify the options, separate them by at least one blank; you can have any number of extra blanks. The order is unimportant. If two contradictory options are specified, the last option specified is accepted and the first ignored.

CMS Record File Examples:

```
KNOWN:   - The file being compiled is FINANCE EXPAND A.
         - A listing of the source file is required.

USE THE FOLLOWING COMMAND:
         CC FINANCE EXPAND (SOURCE

Result:  A listing with the same file name as your source and
         a file type of LISTING is generated.
         When an error occurs, the compiler sends an error
         message to your terminal screen and to your source
```

Figure 23. Specifying Compiler Options under VM/CMS (Example 1)

```
KNOWN:   - The file being compiled is BASEBALL C X.
         - The following disks are to be scanned for
           user include files:
           - V,W,X,Y, and Z (using the LSEARCH option)
           - S,T, and U (using the SEARCH option)
         - The following disks are to be scanned for
           system include files:
           - S,T,U,V,W,X,Y, and Z (using the SEARCH option)

USE THE FOLLOWING COMMAND:
         CC BASEBALL C X (LSEARCH(V) SEARCH(S)

Result:  If the user include file is not found
         on one of the disks specified by the LSEARCH option,
         then the disks specified by the SEARCH option are
         searched for the user include file.

         The disks S through Z are scanned in the standard CMS search
         order for the system include file(s).
```

Figure 24. Specifying Compiler Options under VM/CMS (Example 2)

BFS File Example:

```
KNOWN:   - The file being compiled is myprog.c in the
           current working directory /u/progs.
         - A listing of the source file is required.

USE THE FOLLOWING COMMAND:
         CC ./myprog.c (source

Result:  A listing with the same file name as your source and
         a file extension of lst is generated in your
         current working directory.
```

Figure 25. Specifying Compiler Options for BFS Files

Creating Input Source Files

For CMS record files, the C/C++ compiler accepts both F-format and V-format records. The primary and secondary input can have different formats. For information on mixing formats, see the #pragma sections, margins and sequence in the *C/C++ Language Reference*.

To assist you in migrating existing applications from other operating systems to VM/CMS, file name conversions (described in the following sections) are performed automatically by C/C++ for z/VM These conversions affect file names specified on `#include` preprocessor directives, and in file I/O library functions such as `fopen`. See the *C/C++ Language Reference* for general information on the `#include` directive and the available I/O functions.

Refer to the *z/VM OpenExtensions User's Guide* for complete information on working with BFS files.

Specifying Output Files

The compiler can generate the following kinds of output files:

- Object file
- Listing file
- Preprocessor output

- Events file
- Error message file

When you compile source, you can specify the resultant output file type by using the following compiler options:

Output File Type	Compiler Option
Object File	`OBJECT(filename)`
Listing File	`INLRPT(filename)`
	`LIST(filename)`
	`SOURCE(filename)`
Preprocessor Output	`PPONLY(filename)`
Events File	`EVENTS(filename)`

When you specify any of these compiler options and do not use suboptions to identify the output *filenames*, the compiler generates default output file names based on the type of source file being compiled. Output file names are the same as the source file names. The default output CMS file types and BFS suffixes that the compiler uses are summarized in Table 2.

Table 2. Default CMS File Types and BFS Suffixes for Output Files

Output File	CMS filename Type	BFS filename Suffix
Object File	TEXT	o
Listing File	LISTING	lst
Preprocessor Output	EXPAND	i
Events File	SYSEVENT	err

If you compile source in a CMS record file without specifying output *filenames* in the compiler options, output files are generated on the A disk with the file type shown in Table 2. For example,

```
cc hello c
```

generates object file

```
hello text
```

If you compile source in a BFS file without specifying output *filenames* in the compiler options, output files are generated in the current directory with the suffix shown in Table 2. For example,

```
cc /user/fred/hello.c
```

generates object file

```
./hello.o
```

Events file output is generated using the same file name as the source file and stored on the user's A disk using a file type of `SYSEVENT`.

Error messages are redirected to `stderror` if the `TERMINAL` option is in effect. Error messages can be redirected to a file using the redirection technique, for example:

```
CC A (>ERROR.LOG
```

Valid Input/Output File Types

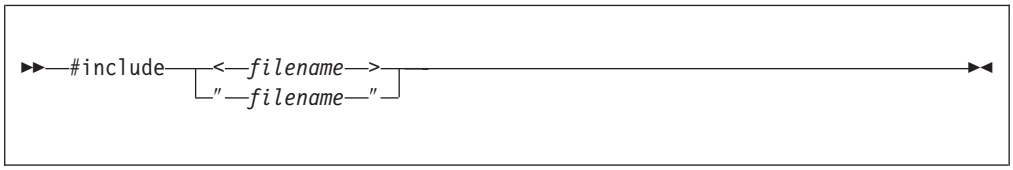
Depending on the type of file that is used as primary input, certain output file types are allowed. The following table describes these input and output file combinations:

Table 3. Valid Combinations of Source and Output File Types

Input Source File	Output File Specified Not In filename Format, for example A B C	Output specified as BFS file, for example a/b/c.o
CMS Native File, for example `A B`	1. If output file exists, overwrites it 2. If output file does not exist, creates the file	1. If the directory does not exist, compilation fails 2. If the directory exists but the file does not exist, creates file 3. If the file exists, overwrites the file.
BFS file, for example `/a/b/d.c`	1. If file exists as a CMS record file, overwrites it 2. If file does not exist, creates output file	1. If the directory does not exist, compilation fails 2. If the directory exists but the file does not exist, creates file 3. If the file exists, overwrites the file.

Using Include Files

The `#include` preprocessor directive allows you to retrieve source statements from secondary input files and incorporate them into your C/C+ program. A description of the `#include` directive is given in the *C/C++ Language Reference*. Its syntax is:



Note: On previous compilers, the double slash at the beginning of *filename* indicated a CMS file. This is not so for C/C++ for z/VM. If you specify it, the CMS minidisks will **NOT** be searched.

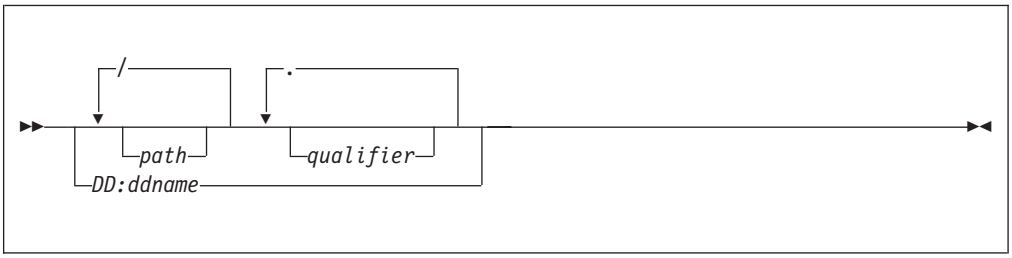
Angle brackets (< >) are used to specify system include files, and double quotation marks (" ") are used to specify user include files.

When you use the `#include` directive, you must be aware of:

1. The file-naming conversions performed by C/C++ for z/VM See "Specifying #include File Names" on page 63 for more information on file name conversions performed by C/C++ for z/VM.
2. The search order used by C/C++ for z/VM to locate the file (known as the *library search sequence*). See "Search Sequences for Include Files" on page 67 for more information on the library search sequence.
3. The area of the input record containing sequence numbers when including files with different record formats. See the *C/C++ Language Reference* for more information on `#pragma sequence`.

Specifying #include File Names: You can use the SEARCH and LSEARCH compiler options to specify search paths for system and user include files. For more information on these options, see “LSEARCH | NOLSEARCH” on page 42 and “SEARCH | NOSEARCH” on page 48.

You can specify a filename using the syntax:



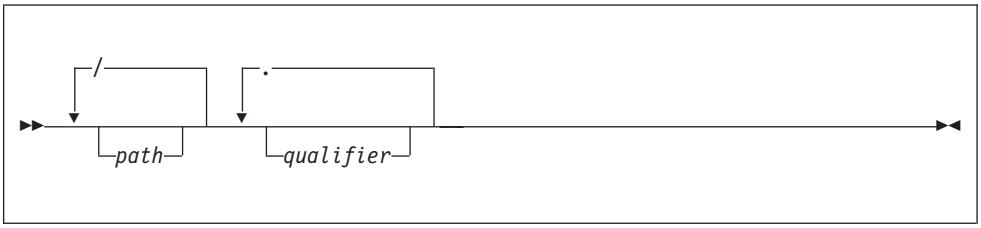
Notes:

1. Absolute CMS file names contain a file mode or are *ddnames*.
2. Absolute BFS file names begin with a leading slash (/) as the first character in *filename*.

Refer to “Determining If filename Is In Absolute Form” on page 64 for more information on absolute file names.

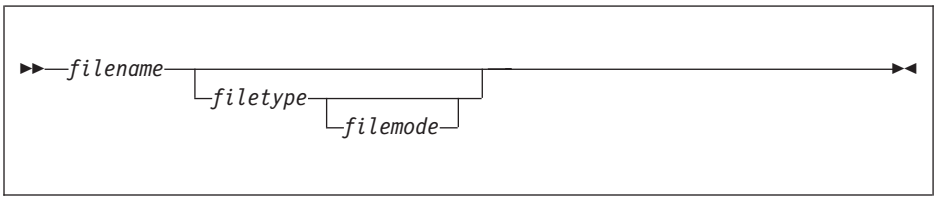
When the compiler performs a library search, *filename* may be treated as a BFS file name or a CMS file name. This depends on whether a CMS library or a BFS directory is being searched. If *filename* is treated as a BFS file name, then no conversions are performed on *filename*. If, on the other hand, *filename* is to be treated as a CMS file name, then the following conversions are performed:

- For the first format:



The compiler performs name conversions in the following order:

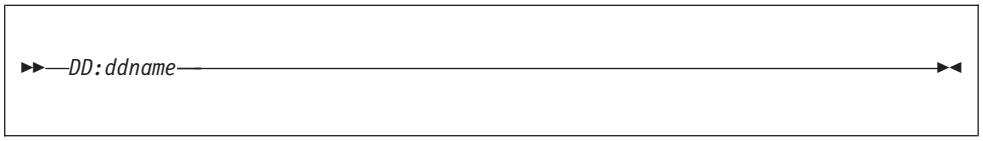
1. All periods (.) are replaced with blank spaces.
2. Characters up to and including the rightmost slash (/) (if any slashes are present) are deleted from the file specification.
3. The remaining file specification must be of the form:



If there are more than three qualifiers, only the first three are used as the file name, file type and file mode, beginning with the leftmost qualifier. The remaining ones are ignored. If you specify the CHECKOUT(PPTRACE) compiler option, a message states what include files the preprocessor is looking for.

4. All file names and file types are truncated to a maximum of eight characters. File modes are truncated to two characters.
5. The file mode must be a valid CMS file mode, or an asterisk (*).
6. If a file mode is not specified, the currently accessed disks are searched in the order described under “Search Sequences for Include Files” on page 67.
7. If a file type is not specified, the default is `H`.

- For the second format:



1. DD: and *ddname* are uppercased and truncated to eight characters.
2. Invalid characters are not converted to at signs(@, hex 7c).

Table 4 gives the original format of the file name as specified on a `#include` directive in a source file, and the actual file name used when C/C++ for z/VM attempts to locate and open the file.

Table 4. Include Directive and Resulting File Names

	`#include` Directive	**Resulting File Name**
	Comments	
1	`#include <stdio.h>`	`STDIO H`
2	`#include <Shoe/Sale/Fall.D>`	`FALL D`
3	`#include "cprog"`	`CPROG H`
4	`#include "utility.h.a"`	`UTILITY H A`
	If the file is not found on disk A, no further search is made.	
5	`#include "DD:MYSYS"`	*file name on* `MYSYS DD`
	The file name associated with the ddname MYSYS will be used.	
6	`#include <DD:PLANLIB>`	*file name on* `PLANLIB DD`
	The file name associated with the ddname PLANLIB will be used.	

Determining If filename Is In Absolute Form

The compiler determines if the *filename* specified in `#include` is in absolute form as follows:

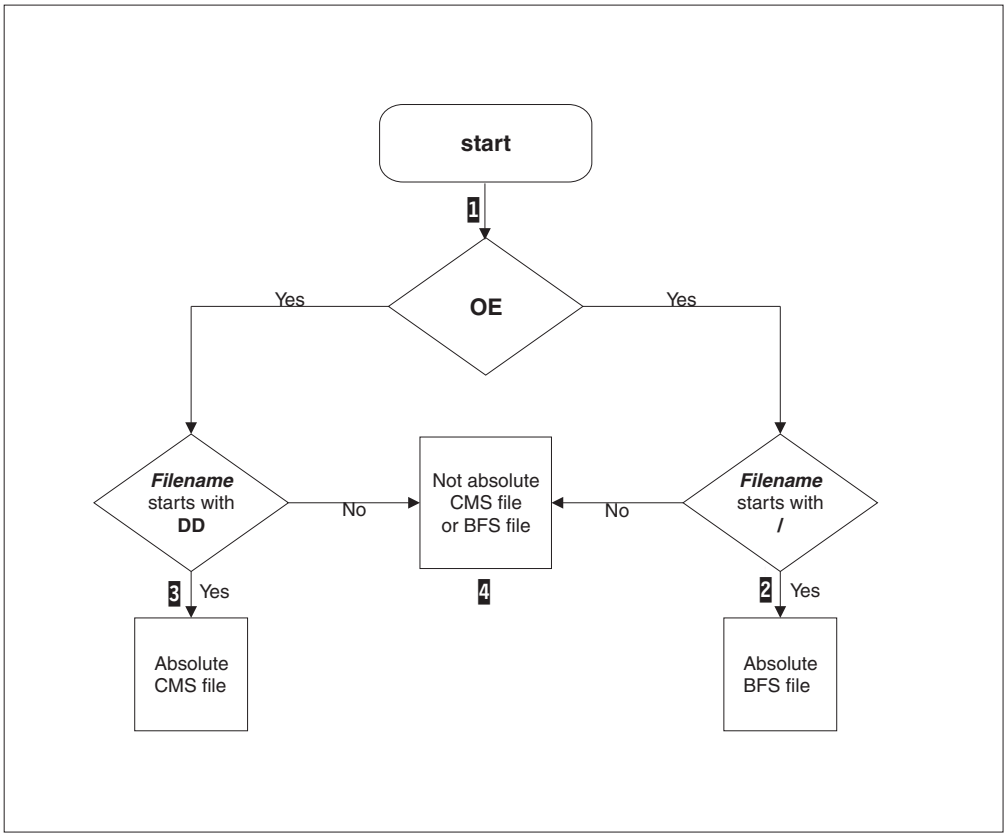


Figure 26. Testing If filename Is In Absolute Form

Notes:

1. The compiler first checks whether the `OE` option is specified.
2. If `OE` is specified, and *filename* starts with a slash (/), then *filename* is in absolute form. The compiler opens the file directly as a BFS file.
3. If `OE` is not specified, and the *ddname* format of the `#include` directive has been used, the compiler uses the file associated with the given `ddname` and directly opens the file. The *ddname* can point to a BFS file. The libraries specified in the `LSEARCH` or `SEARCH` options are ignored.
4. If none of the above conditions are true, then the *filename* is not in absolute format and each `opt` in the `LSEARCH` or `SEARCH` compiler option determines if the file is a BFS or CMS native file.

For example:

```
 Options specified:

  OE

Include Directive:

  #include "apath/afile.h"       NOT absolute, BFS/CMS (no starting slash)
  #include "/apath/afile.h"      absolute BFS, (starts with 1 slash)
  #include "a.b.c"               NOT absolute, BFS/CMS (no starting slash)
  #include "DD:SYSLIB"           NOT absolute, BFS/CMS (no starting slash)
  #include "a.b"                 NOT absolute, BFS/CMS (no starting slash)
```

Using LSEARCH and SEARCH

When the filename in a #include directive is not in absolute format, the *opt*s in SEARCH are used to find system include files and the *opt*s in LSEARCH are used to find user include files. Each *opt* is a library path and its format determines if it is a BFS or CMS path as follows:

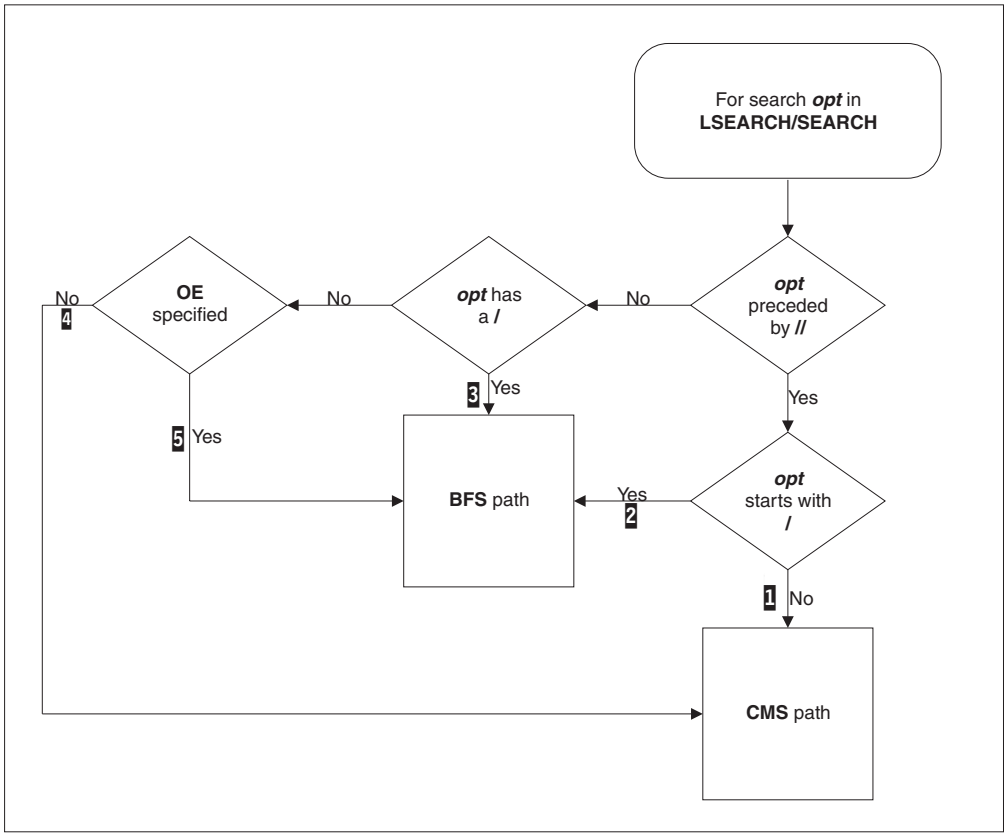


Figure 27. Determining If LSEARCH/SEARCH opt Is BFS Path

Notes:

1. If *opt* is preceded by double slashes (//) and opt does not start with a slash (/), then this path is a CMS path.
2. If *opt* is preceded by double slashes (//) and opt starts with a slash (/), then this path is a BFS path.
3. If *opt* is **not** preceded by double slashes (//) and opt contains a slash (/), then this path is a BFS path.
4. If *opt* is **not** preceded by double slashes (//) and does not contain a slash (/) and `NOOE` is specified, then this path is a non-BFS path.
5. If *opt* is **not** preceded by double slashes (//) and does not contain a slash (/) and `OE` is specified, then this path is a BFS path.

For example:

`SEARCH(./PATH)`	is an explicit BFS path
`OE SEARCH(PATH)`	is treated as a BFS path
`NOOE SEARCH(PATH)`	is treated as a non-BFS path
`OE SEARCH(//PATH)`	is an explicit non-BFS path.

When combining the library with the *filename* specified on the #include directive, it is the form of the library that determines how the include filename is to be transformed. For example:

```
Options specified:

  NOOE LSEARCH(Z, /u/myincs)

Include Directive:

  #include "apath/afile.h"

 Resulting fully qualified include names:

1. AFILE H Z                (Z is non-BFS so filename is treated as non-BFS)
2. /u/myincs/apath/afile.h  (/u/myincs is BFS so filename is treated as BFS)
```

The order of specification of the options on the LSEARCH/SEARCH option is the order that is searched.

If no disk is specified, the file mode A will be added to the end of the LSEARCH/SEARCH option.

See “LSEARCH | NOLSEARCH” on page 42 and “SEARCH | NOSEARCH” on page 48 for more information on these compiler options.

Search Sequences for Include Files

With C/C++ for z/VM, you can specify a search path for locating secondary input files. To specify the search path, you use the LSEARCH and SEARCH compiler options. For details on these compiler options, refer to “LSEARCH | NOLSEARCH” on page 42 and “SEARCH | NOSEARCH” on page 48.

You can search any currently accessed disk or any MACLIB or BFS directory in any order. By default, if there is no LSEARCH or SEARCH option specified, the disks will be searched in the standard VM/CMS order.

If a user include file is not found on the disks or in the MACLIBs or BFS directories specified by the LSEARCH option, the disks and MACLIBs named in the SEARCH option are also scanned in the standard VM/CMS order. Only the disks and MACLIBs specified in the SEARCH option are searched for system include files.

With the NOOE option in effect

Search Sequences for include files are used when the include file is **not** in absolute form. See “Determining If filename Is In Absolute Form” on page 64 for a description of the absolute form of an include file.

If the include filename is not absolute, then the compiler performs the library search as follows:

The search order for **system** include files is:

1. The search order as specified on the SEARCH option, if any
2. The standard CMS disk search, as long as no file mode was specified on the SEARCH option.

The search order for **user** include files is:

1. The search order as specified on the LSEARCH option, if any

2. The standard CMS disk search, as long as no file mode was specified on the LSEARCH option.
3. The search order for system include files.

For example:

```
CC ECONOMY (LSEARCH(X,(*.H)=(LIB(ALPHA.MACLIB))) SEARCH(V)
```

would result in the following search:

Order of Search	For System Include Files	For User Include Files
First	V	X
Second	W	Y
Third	X	Z
Fourth	Y	ALPHA MACLIB (for *.H files)
Fifth	Z	V
Sixth		W

With the OE option in effect

Search Sequences for include files are used when the include file is not in absolute form. See "Determining If filename Is In Absolute Form" on page 64 for a description of the absolute form of an include file.

If the include filename is not absolute then the compiler performs the library search as follows:

- For **system** include files:
  1. The search order as specified on the SEARCH option, if any
  2. The standard CMS disk search, as long as no file mode was specified on the SEARCH option.
- For **user** include files:
  1. If the current source file is a BFS file, the directory of the current source file
  2. The search order as specified on the LSEARCH option, if any
  3. The standard CMS disk search, as long as no file mode was specified on the LSEARCH option
  4. The search order for system include files.

For example, given a file /r/you/cproc containing the following #include directives:

```
#include "/u/usr/header1.h"
#include "common/header2.h"
#include <header3.h>
```

And the following options:

```
OE(/u/crossi/myincs/cproc)
SEARCH(//V, "/new/inc1", "/new/inc2")
LSEARCH("/c/c1", "/c/c2")
```

Then the include files would be searched as follows:

Table 5. Examples of Search Order for OpenExtensions

#include Directive Filename	Files in Search Order
Example 1. This is an absolute pathname, so no search is performed.	

Table 5. Examples of Search Order for OpenExtensions (continued)

#include Directive Filename	**Files in Search Order**
`#include "/u/usr/header1.h"`	1. /u/usr/header1.h
Example 2. This is an OpenExtensions system include file with a relative path name. The search starts with the directory of the parent file or the name specified on the OE option if the parent is the main source file (in this case the parent file is the main source file so the OE suboption is chosen i.e. /u/crossi/myincs).	
″common/header2.h″	1. /u/crossi/myincs/common/header2.h 2. /c/c1/common/header2.h 3. /c/c2/common/header2.h 4. HEADER2 H * 5. HEADER2 H V 6. HEADER2 H W 7. HEADER2 H X 8. HEADER2 H Y 9. HEADER2 H Z 10. /new/inc1/common/header2.h 11. /new/inc2/common/header2.h
Example 3. This is an OpenExtensions system include file with a relative path name. The search follows the order of suboptions of the SEARCH option.	
<header3.h>	1. HEADER3 H V 2. HEADER3 H W 3. HEADER3 H X 4. HEADER3 H Y 5. HEADER3 H Z 6. /new/inc1/common/header3.h 7. /new/inc2/common/header3.h

Binding and Running

This chapter gives an overview of how to bind and run applications using Language Environment under VM/CMS. If you are using OpenExtensions, refer to “z/VM OpenExtensions: Binding and Running a C/C++ Program” on page 84.

Language Environment provides a common Run-Time environment for C, COBOL, and PL/I. For detailed instructions on binding and running existing and new C/C++ for z/VM programs under Language Environment, refer to the *Language Environment Programming Guide*.

The following examples describe how to bind and run a program under VM/CMS in Language Environment. Use the following series of commands to:

1. Bind the C and/or C++ text files.
2. Make the Language Environment library available.
3. Run the load module.

To bind and run a C program:

```
CMOD MYPROG
GLOBAL LOADLIB SCEERUN
MYPROG
```

Figure 28. CMS Commands to Bind and Run a C Program

To bind and run a C++ program:

```
CMOD MYPROG (C++
GLOBAL LOADLIB SCEERUN
MYPROG
```

Figure 29. CMS Commands to Bind and Run a C++ Program

Note: Information on Language Environment is reproduced here for convenience only. For detailed information on Language Environment, please refer to your Language Environment manuals.

Library Routine Considerations

The Language Environment consists of one component that contains all Language Environment enabled languages, such as C, COBOL, and PL/I.

The Language Environment is *dynamic*. That is, many of the functions available in C/C++ for z/VM are not physically stored as a part of your executable program. Instead, only a small portion of code known as a *stub routine* is actually stored with your executable program, and this results in a smaller executable module size. The stub routines contain code that branches to the dynamically loaded Language Environment routine.

Creating an Executable Program

Compilation using the `CC` EXEC produces an object module with file type `TEXT`. Further processing is required to produce an executable module. The simplest way to do this is to use the IBM-supplied `CMOD` EXEC.

The `CMOD` EXEC uses either of the following methods to load one or more object modules (file type `TEXT`) into virtual storage, resolve external references, and create an executable module (file type `MODULE`) on disk:

- Invoke the Binder.
- Invoke the `LOAD` and `GENMOD` CMS commands (and optionally the prelinker).

Note: The Prelinker is not supported for use with C/C++ for z/VM.

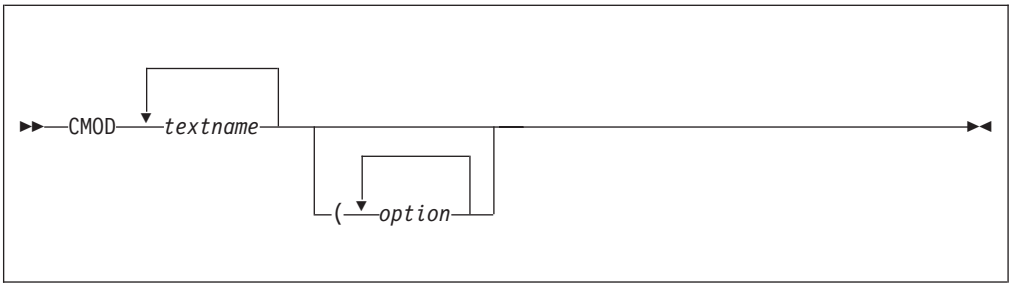
The method used will depend first of all on the options specified on the `CMOD` EXEC. Some `CMOD` options are Binder specific and some are LOAD/GENMOD/Prelinker specific. If any Binder specific options are specified, `CMOD` will use the Binder. If any LOAD/GENMOD/Prelinker specific options are specified, `CMOD` will use LOAD/GENMOD/Prelinker. If both types of options are specified, the type specified first will determine what `CMOD` uses. Warning messages will then be issued for the other type. If no Binder specific or LOAD/GENMOD/Prelinker specific options are specified, `CMOD` will check the value of the `_CNAME` environment variable in the `CENV` group of GLOBALV. If this is set to *CBXFINIT*, which indicates that C/C++ for z/VM

is being used, CMOD will use the Binder. Otherwise, it will use LOAD/GENMOD/Prelinker. Refer to Table 6 on page 72 for more information on CMOD options.

Before using the CMOD EXEC, you should issue a GLOBAL TXTLIB for any user libraries that contain objects that you want to include.

Note: If your application performs long double arithmetic, and runs on a System/370, or on a 370-mode machine under VM XA/ESA, you must have the CMSLIB TXTLIB available.

The general form of the CMOD EXEC is:



where textname is the name of an object module generated by the CC EXEC.

Note: The file containing the function main should be the first file named in the CMOD EXEC. The compiler verifies that main exists by creating a special CSECT that references main.

To specify the name of the executable module, use the MODNAME option of the CMOD EXEC. The CMOD EXEC stores the executable module in a file specified on the MODNAME option. If you do not explicitly name the file in which you want the executable module to be stored, the name of the first object module specified on the CMOD EXEC will be used as the default.

Language Environment Sidedeck Files and TXTLIBs

The CMOD EXEC automatically sets up the appropriate GLOBAL TXTLIB commands and accesses the appropriate sidedeck files to properly create both non-XPLINK and XPLINK C and C++ programs. If you wish to create these without using the CMOD EXEC, that is invoking the Binder yourself, you must also execute the necessary GLOBAL TXTLIB commands prior to invoking the Binder and make the necessary sidedeck files available as primary input to the Binder, as follows:

- for non-XPLINK C programs

```
GLOBAL TXTLIB SCEELKED
Sidedeck file(s): none
```

- for XPLINK C programs

```
GLOBAL TXTLIB SCEEBND2
Sidedeck file(s): CELHS003 CELHS001
```

- for non-XPLINK C++ programs

```
GLOBAL TXTLIB SCEELKED SCEECPP
Sidedeck file(s): C128
```

- for XPLINK C++ programs

```
GLOBAL TXTLIB SCEEBND2
Sidedeck file(s): CELHSCPP CELHS003 CELHS001 C128
```

The sidedeck files are on the Language Environment disk with a filetype of TEXT.

CMOD Options

Table 6. CMOD options

Option	Description
Binder specific options	
BINDOPTS(*options*)	Specifies options for the Binder. These options may be any of the options supported by the Binder. Refer to *DFSMS/MVS Program Management* and/or *z/VM Program Management Binder for CMS* for a complete description of these options.
C++	Specifies that at least one of the text decks is C++. This must be specified for C++ code to be correctly linked.
DLL(*side file name(s)*)	If a side file name is not specified, this just passes the `DYNAM DLL` option to the Binder. It is the same as specifying `BINDOPTS(DYNAM DLL)`, which enables the module for dynamic linking. A definition side file will be produced with the same name as the first text deck name, and a file type of `SYSDEFSD`. If a side file name is specified, the `DYNAM DLL` option is still passed to the Binder, but also the Binder will process the definition side file specified. An 8 character CMS file name is specified. `CMOD` will look for that file name with a file type of `SYSDEFSD`. Multiple names can be specified, separated by blanks. Refer to *DFSMS/MVS Program Management* and/or *z/VM Program Management Binder for CMS* for information about this option. For more information about DLLs (Dynamic Link Libraries), refer to the chapter entitled *Building and Using Dynamic Link Libraries(DLLs)* in the *z/OS C/C++ Programming Guide*, and the chapters entitled *Binding OS/390 C/C++ Programs* and *Binder Processing* in the *z/OS C/C++ User's Guide*.
XPLINK	Specifies that the text deck(s) has been compiled with the `XPLINK` option. Generally speaking, XPLINK text decks cannot be bound with non-XPLINK text decks.
LOAD/GENMOD/Prelinker specific options	
AMODE	Specifies the addressing mode in which the program will be entered in a virtual machine. For a complete description of `AMODE`, refer to the `LOAD` command in the CMS command reference manual.
AUTO|**NOAUTO**	Specifies that your disks are to be searched for `TEXT` files for use in resolving undefined references.
CPLINK(options)	Specifies options for the Prelinker. **Note:** The Prelinker is not supported for use with C/C++ for z/VM.
DUP|NODUP	Specifies that an error message is to be generated if duplicate CSECT names are encountered. If you want to ensure that only one copy of a object module is loaded, use the `NODUP` option.
GENMOD(options)	Passes any options to the `GENMOD` command.
INV|NOINV	Specifies that invalid card images are not to be included in the load map.
LET|**NOLET**	Specifies that all LOAD errors for the load module are to be ignored and an attempt to generate a module will be made.

Table 6. CMOD options (continued)

Option	Description
ORIGIN	Specifies where CMS loads the program. This location must be in the CMS transient area or in any free CMS storage.
RLD|INORLD	Specifies that relocation directory information is to be saved in the load module.
STRI**NOSTR**	Specifies that storage is to be initialized during the generation of the executable module.
RMODE	Specifies where the program is to reside in a virtual machine with greater than 16MB of storage. For a complete description of `RMODE`, refer to the `LOAD` command in the CMS command reference manual.
Common options	
MAP|INOMAP	The specified option is passed to the Binder or the `LOAD` command. For `MAP` (which is the default), the Binder will incorporate a module map into the `SYSPRINT` output (refer to *DFSMS/MVS Program Management* and/or *z/VM Program Management Binder for CMS* for more information); the `LOAD` command will generate a load map file on your A disk with the name `LOAD MAP A`.
MODNAME *modulename*	The default is to generate an executable module having the same file name as the first object module specified, a file type of `MODULE`, and a file mode of `A`. The `MODNAME` option allows you to give a specific name to the executable module. Specify the module name (*modulename*) immediately following the `MODNAME` keyword, and the `CMOD` EXEC creates an executable module named *modulename* `MODULE A`.

Examples

```
KNOWN:   - Only one object module is to be loaded.
         - The object module to be loaded has file name
           PRODUCE, file type TEXT, and file mode A.
         - Default options and file names are to be used.

USE THE FOLLOWING COMMAND:
         CMOD PRODUCE

Note: File type and file mode are not specified on the CMOD EXEC.
```

Figure 30. Example 1 - Using the CMOD EXEC

```
KNOWN:   - The two object modules to be loaded are:
                   - GRAPHING TEXT A
                   - TRIG TEXT A
         - GRAPHING TEXT A contains the main().
         - A load module map is to be generated.
         - The load module produced is to be called MATH MODULE A.

USE THE FOLLOWING COMMAND:
         CMOD GRAPHING TRIG (MODNAME(MATH) MAP
```

Figure 31. Example 2 - Using the CMOD EXEC

```
KNOWN:   - Only one object module is to be loaded.
         - The object module to be loaded has file name
           DLLA07, file type TEXT, and file mode A.
         - The load module is to be a DLL
         - Default options and file names are to be used.

USE THE FOLLOWING COMMAND:
         CMOD DLLA07 (DLL
Note: A definition side file with name DLLA07 and type SYSDEFSD
      will be produced on the A disk.
```

Figure 32. Example 3 - Using the CMOD EXEC

```
KNOWN:   - Only one object module is to be loaded.
         - The object module to be loaded has file name
           C955A07, file type TEXT, and file mode A.
         - C955A07 calls functions in the DLLA07 DLL.
         - There is a definition side file called DLLA07 SYSDEFSD.
         - Default options and file names are to be used.

USE THE FOLLOWING COMMAND:
         CMOD C955A07 (DLL(DLLA07)
```

Figure 33. Example 4 - Using the CMOD EXEC

Using the LOAD and GENMOD Commands

Note: This method of creating an executable program cannot be used for C text files that were compiled with either the LONGNAME or RENT options, or for C++ text files. These text files need to be processed by the binder to resolve writable static references and/or map long internal names to short external names.

The loader can also be invoked under VM/CMS by using the LOAD command processor. For complete details on the LOAD, INCLUDE, and GENMOD commands, refer to the list of publications given under “z/VM Base Publications” on page 163.

Compilation using the CC command produces an object module with the file type TEXT. To run the program, you must load the object module to form a load module before you can execute it.

The LOAD command invokes the loader, which loads one or more object modules and creates an executable module in virtual storage.

Note that the object modules you are loading with the LOAD command must have a file type of TEXT. If you are loading several object modules, the file names must be separated by at least one blank. You can also specify load options following the input file names. If you want to specify more than one load option, the options must be separated by blanks.

Default options for the LOAD command are described in the *z/VM CMS Command and Utility Reference*, SC24-6010.

The general form of the LOAD command is:

```
LOAD filename1 filename2 ... filenameN (options
```

Note: If the main program is C, you should include `RESET CEESTART` under the options for the `LOAD` command. The object module that contains the main must be the first one specified.

To store the executable module that was created by the loader in a file, use the `GENMOD` command. The `GENMOD` command will take a copy of the executable module in virtual storage and store it with the file name specified on the `GENMOD` command. Only the file name is required on the `GENMOD` command.

The general form of the `GENMOD` command is:

```
GENMOD filename (options
```

Notes:

1. If you specify a file type, it must be `MODULE`.
2. If the main program is C, then under the options for the `GENMOD` command, you should include `FROM CEESTART`.

If you do not explicitly name the file in which you want the load module to be stored, the `GENMOD` command processor defaults to the name of the first entry point in the load map. The following example shows you how to override the default to produce a load module with a user-specified file name.

```
KNOWN:  - Three object modules are to be loaded:
                - IMPORTS TEXT A
                - EXPORTS TEXT A
                - FORMULA TEXT A
        - The load module is to be called GNP MODULE A.
        - The main procedure is in IMPORTS.
        - Default options are to be used.

USE THE FOLLOWING COMMAND:
        GLOBAL TXTLIB SCEELKED CMSLIB
        LOAD IMPORTS EXPORTS FORMULA (RESET CEESTART
        GENMOD GNP (FROM CEESTART
```

Figure 34. Using the LOAD and GENMOD commands

For more information on linking modules for interlanguage calls, see “Linking Modules for Interlanguage Calls” on page 77.

Using the BIND Command

The `BIND` command invokes the *z/VM Program Management Binder for CMS*, which encompasses the functionality of the Prelinker, the `LKED` command, and the `LOAD` and `GENMOD` commands. In addition, it supports the OS/390 Program Object format which is required for some compiler options such as `XPLINK`.

The Prelinker is not supported for use with C/C++ for z/VM. Any C programs which use the `RENT` or `LONGNAME` options, or any C++ programs must use the Binder to create an executable module.

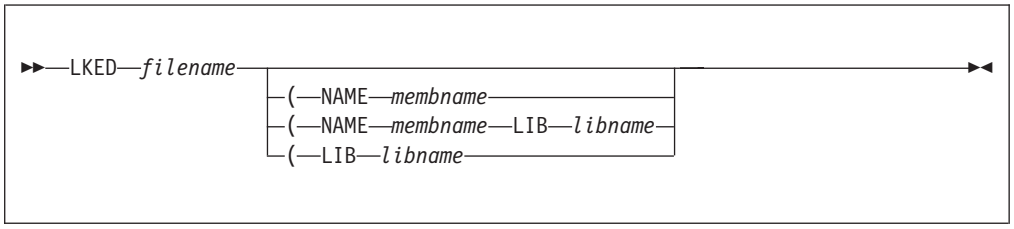
For more information about the *z/VM Program Management Binder for CMS* see the *z/VM Program Management Binder for CMS* manual and the chapters entitled *Binding OS/390 C/C++ Programs* and *Binder Processing* in the *z/OS C/C++ User's Guide*.

Using the LKED Command

The LKED command is used to create a member of a CMS load library. CMS load libraries, like text libraries, are in CMS partitioned data set formats. Text libraries contain applications that contain unresolved external references to other routines. Load libraries, on the other hand, contain applications with external references that have already been resolved, thus saving overhead every time the application is loaded.

Your TEXT file is input to the LKED command. If your application calls a subroutine with object code stored as a separate TEXT file or as a member of a text library, you must define the files that contain the subroutines used by your application with a FILEDEF command.

After you issue the appropriate FILEDEF commands, issue the LKED command as follows:



filename
Name of the TEXT file that contains your object code and/or linkage editor control cards.

NAME *membname*
Member name to be used for the load module that is created.

LIB *libname*
Name of the LOADLIB file where the resulting load module is placed.

The following example causes the automatic call library to search SCEELKED to resolve external references, creates a load library member named PROGRAM1, and stores it in a CMS load library with the name USERLOAD.

```
FILEDEF SYSLIB DISK SCEELKED TXTLIB E
LKED PROGRAM1 (NAME PROGRAM1 LIB USERLOAD
```

Using FILEDEF to Define Input and Output Files

If your program opens files by ddname (`fopen("DD:INFILE","r");`), you must issue a corresponding FILEDEF prior to executing your program. The FILEDEF command relates the ddname of the input or output file specified in your application with an I/O device. For example, if PROGRAM1 contains a ddname of an input file stored on your A disk as MYDATA INPUT, issue the following command:

```
FILEDEF infile DISK MYDATA INPUT A
```

where infile is the ddname of the input file specified in PROGRAM1.

Preparing a Reentrant Program

Reentrancy allows more than one user to share a single copy of a load module or to repeatedly use a load module without reloading it.

Reentrant programs can be categorized by their reentrancy type as follows:

- Natural reentrancy - Programs that contain no writable static and do not require additional processing to make them reentrant.
- Constructed reentrancy - Programs that contain writable static and require additional processing to make them reentrant.

Note: All C++ programs use constructed reentrancy. They cannot be compiled with the `NORENT` option.

Writable static is storage that changes and is maintained throughout program execution. It is made up of:

- All program variables with the static storage class.
- All program variables receiving the extern storage class
- All writable strings

Note: If your program contains no writable strings and none of your static or extern variables are updated in your application (that is, they are read only), your program is naturally reentrant.

To generate a reentrant load module, you must follow these steps:

1. If your program contains writable static, you must compile all your C source files using the `RENT` compiler option.

   If you are unsure about whether your program contains writable static, compile it with the `RENT` option. Invoking the Binder with the `MAP` option and the object module as input produces a module map. Any writable static data in the object module appears in the writable static section of the map.
2. Use the Binder to combine all input object modules into a single output object module.
3. Optionally, do one of the following:
   - Have your system programmer link/install your program into an NSS area of the system.
   - Install your program as a nucleus extension by using the VM/CMS `NUCXLOAD` command.

   Refer to the publications given under “z/VM Base Publications” on page 163 for details.

   You do not need to install your program to run but if you do not, you will not gain all the benefits of reentrancy.

Linking Modules for Interlanguage Calls

For information on link-editing modules for interlanguage calls, refer to the *Language Environment Programming Guide*.

Running a Program

Once you have compiled and loaded your C/C++ program, you can run it one of two ways:

1. Using the *file name* of the load module. For example:

   ```
   TESTRUN
   ```
2. Using the `START` command immediately after a `LOAD` or `LOADMOD` command. For example:

```
LOADMOD TESTRUN
START
```

Making the Run-Time Libraries Available for Execution

The Language Environment must be available at Run-Time for your application to use the dynamic library routines. The following sections describe how to make these libraries available to your programs.

Making the Language Environment Library Available for VM/CMS

The C specific portions of the Language Environment are in modules `CEEEV003`, `CELHV003` and `EDCZ24`. `CEEEV003` is the main C run-time module for non-XPLINK programs, `CELHV003` is the main C run-time module for XPLINK programs, and `EDCZ24` is the I/O routine module.

These can be loaded as nucleus extensions, Named Saved Segments (NSS) or directly from the LE minidisk or directory. Nucleus extensions and NSSs offer improved performance.

Other portions of the Language Environment dynamic environment are on the Languaqe Environment minidisk or directory in the form of separate modules and the `SCEERUN` loadlib. The modules can also be loaded as nucleus extensions or NSSs. The loadlib needs to be accessed with the `GLOBAL LOADLIB` command. For example:

```
GLOBAL LOADLIB SCEERUN
```

Search Sequence for Library Files

The search order for the library files is:

1. Nucleus extension
2. Named Saved Segment
3. LOADLIB

For best performance, the library should be loaded as a nucleus extension.

Specifying Run-Time Options

Each time a C/C++ program is run, values must be established for a set of C/C++ Run-Time options. These options affect many of the properties of a C/C++ program's execution, including its performance, its error handling characteristics, and its production of debugging and tuning information.

If the `EXECOPS` Run-Time option is in effect and if you want to specify additional Run-Time options on the command line, specify the options, followed by a slash (/), followed by the parameters you want to pass to the `main` function.

If the `NOEXECOPS` Run-Time option is in effect, any arguments and options that you specify on the command line (including the slash, if present) are passed as arguments to the `main` function. Run-Time options are described in "Specifying Run-Time Options" on page 52.

Each time the program is run, the default Run-Time options selected during C/C++ installation apply unless overridden by options specified in a `#pragma runopts` directive in your source program or by command line options specified at the time of program execution.

Run-Time options are specified using the runopts pragma or in the *parameter-string* on the command line when you invoke your C/C++ program. The *parameter-string* contains two fields separated by a slash(/), and takes the form:

[*Run-Time options*/][*parameter string*]

The first field is passed to the program initialization routine as a Run-Time option list; the second passes to the main function. If you do not specify any Run-Time options but want to pass arguments, you must precede the arguments with a slash.

The following example shows you how to specify Run-Time options and pass arguments when you invoke your program under VM/CMS.

```
KNOWN:   - The load module to be executed is called SURVEY MODULE A.
         - You want to pass the words THIS IS A TEST to
           the program.
         - The messages generated by the Run-Time library
           will be received in Japanese.

USE THE FOLLOWING COMMAND:
         SURVEY LANG(JA)/THIS IS A TEST
```

Figure 35. Running under CMS

Message Handling

By default, all C/C++ for z/VM programs (including the compiler) set emsg off so that VM/CMS messages generated during normal execution of C library functions are not output to the terminal along with stdout and stderr. The C system function restores the emsg setting, issues the given command in the system call, and sets emsg off again.

Use the setenv() function to set emsg via the C environment variable _EDC_KEEP_EMSG, as follows:

```
setenv("_EDC_KEEP_EMSG","Y",1);
```

This environment variable restores the emsg setting to its value prior to the execution of the C program, and keeps that value while the program is running.

There are 4 ways to get C/C++ for z/VM to leave the emsg setting on. This allows any messages produced by VM/CMS during execution of your program to be displayed.

- Issue the CMS command GLOBALV SELECT CENV SET _EDC_KEEP_EMSG Y.
- Modify the user exit CEEBINT to issue a setenv("_EDC_KEEP_EMSG","Y",1) and link this with your executable module.
- Create a variable length file with a line of the following format. (Spaces are not permitted.)

  ```
  _EDC_KEEP_EMSG=Y
  ```

  Create a FILEDEF EDCENV DISK fn ft fm for the same file. During initialization of the root main program, C/C++ for z/VM opens the file associated with the ddname EDCENV and sets the appropriate environment variables.
- Issue a setenv("_EDC_KEEP_EMSG","Y",1) in your program. This restores emsg to the value in effect when your program was invoked.

The environment variable may be set any time in a C program, or may be set in the Run-Time user exit `CEEBINT`.

If the `emsg` setting is changed via a `system()` call once `_EDC_KEEP_EMSG` has been set, then the new `emsg` setting will be maintained even after the C program terminates.

For additional information on the `setenv()` library function, refer to the *C/C++ for z/VM Run-Time Library Reference*. For more information on environment variables, refer to that section of the *z/OS C/C++ Programming Guide*. Additional information on Run-Time user exits in C/C++ for z/VM is also available in the *z/OS C/C++ Programming Guide*.

Chapter 4. z/VM OpenExtensions: Compiling, Binding and Running

z/VM OpenExtensions: Compiling a C/C++ Program

This chapter describes how to compile your program using the C/C++ for z/VM compiler with the `c89` and `cxx` utilities under z/VM OpenExtensions. For detailed information regarding the `c89` utility options and operands, refer to the *z/VM OpenExtensions Command Reference*.

Compiling with c89/cxx

An OpenExtensions C/C++ for z/VM application program with source code in BFS files or CMS native files must be compiled to create output object files residing either in BFS files or VM record files.

Application source code can be compiled and built at one time or compiled and then bound at another time with other application source files or compiled objects.

To compile and build an OpenExtensions application program from the z/VM OpenExtensions shell, use the `c89` or `cxx` utility. Use the `c89` utility to compile and build C only applications. Use the `cxx` utility to compile and build C/C++ applications.

Note: All further references to `c89` also apply to `cxx` unless otherwise specified.

The syntax for `cxx` is the same as for `c89`. The syntax is:

```
c89 [-options ...] [file.c ...] [file.a ...] [file.o ...] [-l libname]
```

where:

options are one of the `c89` options described in "c89 Selectable Compiler Settings" on page 50.

file.c is the source file.

file.o is the object file.

file.a is the archive file.

libname is the archive library.

Note: You can compile and build application program source and objects from within the shell or directly from CMS using the OpenExtensions `c89` utility. If you use `c89`, you must keep track of and maintain all the source and object files for the application program. However, you can take advantage of the `make` utility and create makefiles to maintain your z/VM OpenExtensions application source and object files automatically when you update individual modules. The `make` utility runs `c89` for you. However, `make` must be run from the shell.

For more information on using the make utility, see “ar and make Utlities” on page 115 and OpenExtensions for *VM/ESA: Advanced Application Programming Tools*.

To compile source files without binding them, enter the `c89` command with the `-c` option to create object file output. Use the `-o` option to specify placement of the application program executable file to be generated. The placement of the intermediate object file output depends on the location of the source file:

- If the C/C++ source module is a BFS file, the object file is created in the working directory.
- If the C/C++ source module is a CMS native file, the object file is created as a CMS native file. The object file is placed in the CMS minidisk or SFS directory accessed as file mode A.

  For example, if the C/C++ source is in a minidisk file named `USERSRC C B`, the object is placed in the file `USERSRC TEXT A`. Since the CMS file ID is always converted to uppercase, you can specify it in lowercase or mixed case.
- Compiling application source to produce only object files.
  - To compile C/C++ source to create the default object file `usersource.o` in your working BFS directory, specify:

    ```
    c89 -c usersource.c
    ```
  - To compile C/C++ source to create an object file as a file on the A disk, specify:

    ```
    c89 -c //approg.c
    ```
- Compiling and binding application source to produce an application executable file.
  - To compile an application source file to create the object file `usersource.o` in the BFS working directory and the executable file `mymod.out` in the `/app/bin` directory, specify:

    ```
    c89 -o /app/bin/mymod.out usersource.c
    ```
  - To compile the C source file `MAINBAL C` on the B disk and build it to produce the application executable file `/u/parker/myappls/bin/mainbal.out`, specify:

    ```
    c89 -o /u/parker/myappls/bin/mainbal.out //mainbal.c.b
    ```

Compiler Selection

By default, the `c89` utility will call the IBM C for VM/ESA compiler. To make it call the C/C++ for z/VM compiler, set the `_CNAME` environment variable in the `CENV` group of GLOBALV to *CBXFINIT*. This will also cause `c89` to call the Binder instead of the Prelinker.

The `cxx` utility ignores the setting of the `_CNAME` environment variable and always calls the C/C++ for z/VM compiler.

Compiling and Building in One Step with c89/cxx

To compile and build an OpenExtensions C/C++ for z/VM application program in one step to produce an executable file, specify the `c89/cxx` utility without specifying the `-c` option.

Note: To compile source files without building them, use the `c89 -c` option. This will create object files only.

You can use the `-o` option with the command to specify the name and location of the application program executable file to be created.

- To compile and build an application program source file to create the default executable file `a.out` in the BFS working directory, specify:

  ```
  c89 usersource.c
  ```

- To compile and build an application source file to create the `mymod.out` executable file in your `/app/bin directory`, specify:

  ```
  c89 -o /app/bin/mymod.out usersource.c
  ```

- To compile and build several application source files to create the `mymod.out` executable file in your `/app/bin directory`, specify:

  ```
  c89 -o /app/bin/mymod.out usersource.c ottrsrc.c //pwapp.c
  ```

- To compile and build an application source file to create the `MYLOADMD` module file on your A disk specify:

  ```
  c89 -o //myloadmd.module usersource.c
  ```

- To compile and build an application source file with a previously compiled object file to create the executable file `zinfo` in your `/approg/lib` BFS directory, specify:

  ```
  c89 -o /approg/lib/zinfo usersource.c existobj.o //pwapp.c
  ```

Using the make Utility

You can use the z/VM OpenExtensions shell `make` utility to control your OpenExtensions C/C++ for z/VM application's parts. The `make` utility calls the `c89` utility by default to compile and bind the programs specified in the previously created makefile.

The `/etc/startup.mk` file contains the `make` default rules.

For example, if you have the file `/u/jake/appwrk/makefile.c` that contains the dependencies for your C application program `primappl` and you make changes to the source file `subordpgm.c`, you can recompile the application by entering:

```
cd appwrk
make -f makefile.c
```

The result is the same as if you had entered:

```
c89 -O -o primappl ./appwrk/subordpgm.c
```

Note: The z/VM OpenExtensions make utility requires that any application program source files to be ″maintained″ through use of a makefile reside in BFS files. To compile and build C/C++ source files that are in CMS native files you must use the `c89` utility directly.

See *z/VM OpenExtensions Command Reference* for a description of the `make` utility. For a detailed discussion on how to create and use makefiles to manage application parts, see *z/VM OpenExtensions Advanced Application Programming Tools*.

z/VM OpenExtensions: Binding and Running a C/C++ Program

This chapter describes how to bind and run C/C++ for z/VM programs under z/VM OpenExtensions.

The interfaces to the CMS module build facilities for OpenExtensions C/C++ for z/VM applications are the z/VM OpenExtensions `c89` utility and the `cxx` utlity. You can use `c89/cxx` to compile and build an OpenExtensions C/C++ application program in one step or bind application object files after compilation. For more information on compiling with the `c89/cxx` utility, refer to "z/VM OpenExtensions: Compiling a C/C++ Program" on page 81.

Note: All references to `c89` in the following sections also apply to `cxx` unless otherwise specified.

Using the c89 Utility to Bind and Create Executable Files

To bind an OpenExtensions C/C++ for z/VM application program's object files to produce an executable file, specify the `c89` utility and pass it object files (*file*`.o` BFS files or CMS native files). The `c89` utility recognizes that these are object files produced by previous C/C++ compilations and does not invoke the compiler for them.

To compile source files without binding them, use the `c89 -c` option to create object files only.

You can use the `-o` option with the command to specify the name and location of the application program executable file to be created.

- To bind an application program object file to create the default executable file `a.out` in the working directory, specify:

  ```
  c89 usersource.o
  ```

- To bind an application object file to create the `mymod.out` executable file in the `app/bin` directory, relative to your working directory, specify:

  ```
  c89 -o ./app/bin/mymod.out usersource.o
  ```

  where `usersource.o` is the object file created by compilation with `c89`.

- To bind several application object files to create the `mymod.out` executable file in the `app/bin` directory, relative to your working directory, specify:

  ```
  c89 -o ./app/bin/mymod.out usersrc.o othersrc.o
  ```

- To bind an application object file to create the `MYLOADMD` module file on the A disk specify:

  ```
  c89 -o //myloadmd.module usersource.o
  ```

- To compile and bind an application source file with several previously compiled object files to create the executable file `zinfo` in the `approg/lib` subdirectory, relative to your working directory, specify:

  ```
  c89 -o ./approg/lib/zinfo usersrc.c existobj.o //pgmobj.text
  ```

c89 Binder Options

The `c89` and `cxx` utilities specify default values for some Binder options. They also pass Binder options by using the `-W` option. For more information on using the `c89` options, see "Compiler Options under OpenExtensions" on page 49.

Binder Options

c89 uses the following Binder options, all of which can be overridden using the -W option:

```
CASE MIXED TERM DISK
```

cxx uses the following Binder options:

```
CASE MIXED TERM DISK RENT DYNAM DLL
```

The following example shows how to use the -W option to pass a Binder option.

```
c89 -Wb,b,map,case,upper hello.c
```

Refer to *DFSMS/MVS Program Management* and/or *z/VM Program Management Binder for CMS* for more information about Binder options.

Specifying Run-Time Options under OpenExtensions

If you have an OpenExtensions C/C++ for z/VM application program executable file in the byte file system (BFS), you cannot run the executable file by simply entering its name on the CMS command line, as you would a traditional CMS application program. Instead, you can execute the application by specifying its name on the CMS command OPENVM RUN. However, OPENVM RUN does not support passing of Run-Time options to the application.

Run-Time options, needed for the OpenExtensions application program residing in the BFS, can be passed from a `#pragma runopts` preprocessor directive at compile time. When Run-Time options are specified in this way a `CEEUOPT` control section (CSECT) is created and is linked with the application program by the `c89` utility. Because only one `CEEUOPT` CSECT can be linked with an application program, you should code a `#pragma runopts` directive in the compilation unit for the `main()` function. For more information about `#pragma runopts`, refer to “Run-Time Options Using the Language Environment” on page 52.

Note: Also, you can create a `CEEUOPT CSECT` as a separate step using the `CEEXOPT` macro and bind the CSECT with the application program object files using `c89`.

Running under z/VM OpenExtensions

This section discusses how to run your OpenExtensions VM C/C++ for z/VM application program executable files on the z/VM system.

z/VM OpenExtensions Application Program Environments

z/VM OpenExtensions supports the following environments, from which you can run your OpenExtensions C/C++ for z/VM application programs:

- z/VM OpenExtensions shell
- CMS

Placing a CMS Application Program Load Module in the File System

If you have an OpenExtensions C/C++ for z/VM application program executable file as a CMS native file and want to place it in the BFS, you can use the following z/VM OpenExtensions CMS commands to copy the file into a BFS file:

- `OPENVM PUTBFS`

For a description of this command, see the *z/VM OpenExtensions Command Reference*. For examples of using these commands to copy CMS files into BFS, see the *z/VM OpenExtensions User's Guide*.

Running a CMS Module from the z/VM OpenExtensions Shell

If your OpenExtensions C/C++ for z/VM program is a CMS module file on a minidisk or in the shared file system, you can invoke it from the shell by using the `cms` command. For example, to run `PROG1 MODULE A`, execute the following command:

```
cms prog1
```

If you want to make the module file transparent to the shell, you need to create an external link in the BFS that points to the file. For example, to run `PROG1 MODULE A`, you can create a file in the BFS that represents the module by using the following command:

```
openvm create extlink /u/mydir/prog1 cmsexec PROG1 MODULE A
```

You can run the module transparently from the shell by using the following command:

```
prog1
```

See the *z/VM OpenExtensions Command Reference* for more information on creating external links.

Running an OpenExtensions C/C++ for z/VM Application Executable File from the z/VM OpenExtensions Shell

If the application executable file is a BFS file, you must either run it from the shell interactively or invoke it indirectly through the CMS command OPENVM RUN.

Issuing the Executable Filename from the Shell: Before a BFS program can be run in the z/VM OpenExtensions shell, it must be given the appropriate mode authority for a user or group of users to run it. You can update the mode authority for an executable program file by using the `chmod` command. See the *z/VM OpenExtensions Command Reference* for the format and description of `chmod`.

After you have updated the mode authority, enter the program name from the z/VM OpenExtensions shell command line. For example, if you want to run the program `datcrnch` from your working directory, you have the directory where the program resides defined in your search path, and you are authorized to run the program, enter:

```
datcrnch
```

Issuing a Setup Shell Script Filename from the Shell: To run an z/VM OpenExtensions shell script that sets up an OpenExtensions VM executable file and then runs the program, give the appropriate mode authority for a user or group of users to run it. You can update the mode authority for a shell script file by using the `chmod` command. See the *z/VM OpenExtensions Command Reference* for the format and description of `chmod`. After mode authority has been given, enter the script filename from the z/VM OpenExtensions shell command line.

Chapter 5. C/C++ for z/VM Utilities

Object Library Utility

This chapter describes how to use the Object Library Utility to update libraries of object files. On VM/CMS, a library is a text library (`TXTLIB`) with object files as members.

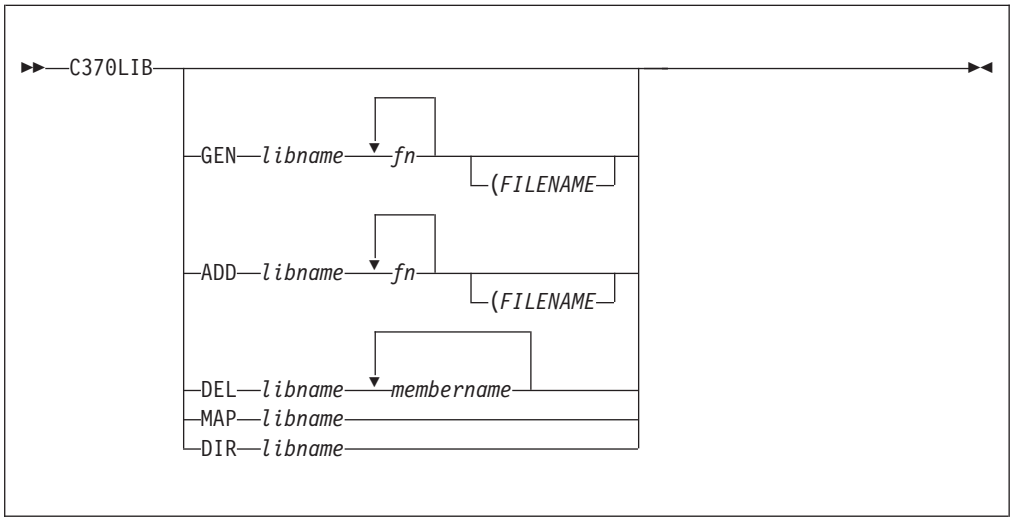
Object libraries provide convenient packaging of object files. With the Object Library Utility, a library can contain objects files compiled with long names, short names, writable static data, or XPLINK. The Object Library Utility stores source member symbol information with different attributes. This information is stored in two special members of the library, the Basic Directory Member (`@@DC370$`) and the Enhanced Directory Member (`@@DC390$`). Both are referred to in this chapter as the `C370LIB` directory.

Note: The TXTLIB command under VM/CMS also creates object libraries but you cannot include external names longer than 8 characters. The syntax for the Object Library Utility is similar to the TXTLIB command.

Commands to add object files to a library, to delete object files from a library, or to build the C370LIB directory for a library are available. Use the DIR command to build the C370LIB directory for a library of object files. Use the MAP command to list the contents of the C370LIB directory.

Creating an Object Library under VM/CMS

You use the C370LIB EXEC to create an object library.



where:

GEN Creates a TXTLIB on your A disk. If a TXTLIB with the same name already exists, it is replaced.

ADD Adds TEXT files as members to an existing TXTLIB on a read/write disk. No checking is done for duplicate names, entry points, or CSECTs.

DEL Deletes members from a TXTLIB on a read/write disk and compresses the TXTLIB to remove unused space. If more than one member exists with the same name, only the first entry is deleted.

MAP Lists the names (entry points) of TXTLIB members.

MAP produces a file, libname MAP, on your A disk. See “Object Library Utility Map” on page 90 for more information on the map.

DIR Builds the C370LIB directory. The C370LIB directory contains the names (entry points) of library members.

The DIR function is only necessary if TEXT files were previously added or deleted from the TXTLIB without using C370LIB.

libname Specifies the file name of a file with a file type of TXTLIB, which can be one of:

- A library that is to be created or listed
- A library to which members are to be added

- A library from which members are to be deleted
- A library for which a C370LIB directory is to be built

fn Specifies one or more names of files with file types of TEXT, that you want to add to a TXTLIB.

membername Specifies one or more names of TXTLIB members that you want to delete.

FILENAME Indicates that all the file names specified (fn ...) will be used as the member names for their respective entries in the TXTLIB file.

C370LIB must be used to update a TXTLIB with TEXT files produced by compiling C programs with the LONGNAME option, or compiling C++ programs. The VM/CMS TXTLIB command cannot be used to do this directly, and an error can result if this is attempted.

When a TEXT file is added to a library, its member name is selected according to the following hierarchy:

1. From the file name, if the FILENAME option is specified
2. From the NAME control statement, if present, in the TEXT file
3. From the file name.

The CMS TXTLIB command GEN, ADD, and DEL functions are used as part of the C370LIB GEN, ADD and DEL functions. Thus, any TXTLIB restrictions apply also to C370LIB unless otherwise stated. See the appropriate manual listed in "z/VM Base Publications" on page 163 for information on the TXTLIB command.

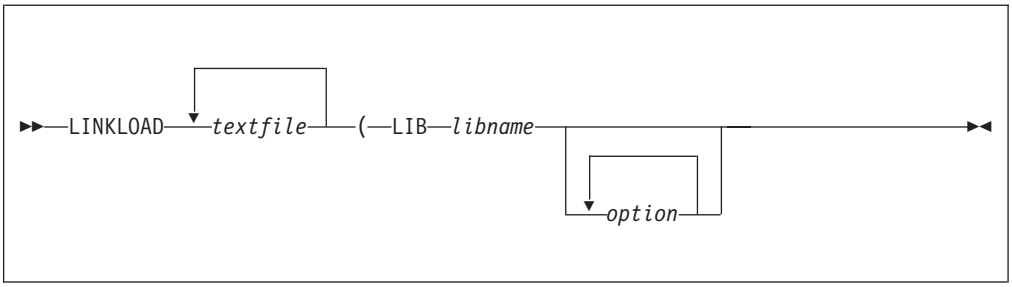
Members must be deleted by their member name. Any attempt to delete a member using a name other than the member name will result in a warning message.

In the following example, the C programs SUB1 C and SUB2 C are compiled with the LONGNAME option. The function library, SUBLIB TXTLIB A, is created with SUB1 TEXT using the GEN command of C370LIB, the Object Library Utility. SUB2 TEXT is added to the library using the ADD command.

```
CC SUB1 (LO
CC SUB2 (LO
C370LIB GEN SUBLIB SUB1
C370LIB ADD SUBLIB SUB2
```

LINKLOAD EXEC

The IBM-supplied LINKLOAD EXEC will generate a fetchable member of a VM/CMS load library.



where:

textfile Specifies one or more names of the input text files. The file type of

the object files must be TEXT, and the source programs must have contained a #pragma linkage(name,FETCHABLE) preprocessor directive. Note that you do not specify the file type or the file mode when using the LINKLOAD EXEC.

libname Specifies the name of the library where the load member is to be stored.

option Specifies the options you want to apply when you are generating the fetchable load library member.

MBR is a keyword used to indicate that the next argument, *memname*, is the name of the member within the load library that is to be generated. If you do not specify a member name, the name of the text file containing the fetchable code is used.

CPLINK *options* allows you to pass options to the Prelinker. CPLINK is called if it is required by the text file or if a cplink option is given.

Note: The Prelinker is not supported for use with C/C++ for z/VM.

LKED is a keyword used to indicate that the options following it are to be passed to LKED. If you do not use this option, default options are used.

Only one of the following options can be specified on a given invocation of LINKLOAD:

ADD indicates that the load member generated by the LINKLOAD EXEC is to be added to the load library. If a member by the same name already exists, the new member will not be added.

REPLACE indicates that the load member generated by the LINKLOAD EXEC is to replace the member having the same name in the load library. If a member by the same name does not exist, the new member is added.

NEW indicates that, if a load library of the given name exists then it is erased, and a new load library containing the new member is created.

Object Library Utility Map

The Object Library Utility produces a listing for a given library when the MAP command is specified. The listing contains information on each member of the library.

```
=========================================================================
1 |                      Object Library Utility Map                       |
  |                                                                       |
  |C370LIB:5647A01 V2 R10 M0 IBM Language Environment 2001/01/03 15:10:51|
=========================================================================

   Library Name: FRANK.A.OBJLIB

  *-----------------------------------------------------------------------*
2 *  Member Name:    CGOFF                       (P) 2001/00/00 13:51:23 *
  *                                                   5647A01    V2 R10   *
  *-----------------------------------------------------------------------*

3    User Comment:
  AGGRCOPY(NOOVERLAP) NOALIAS ANSIALIAS ARCH(2) ARGPARSE NOCOMPACT
  NOCOMPRESS NOCONVLIT CSECT() NODLL(NOCALLBACKANY) EXECOPS NOEXPORTALL
  FLOAT(HEX, FOLD, NOAFP) GOFF NOGONUMBER NOHWOPTS NOIGNERRNO NOINITAUTO
```

```
NOINLINE NOIPA LANGLVL(*EXTENDED) NOLIBANSI NOLOCALE LONGNAME
MAXMEM(2097152) OPTIMIZE(0) PLIST(HOST) REDIR RENT NOROCONST NOROSTRING
NOSERVICE SPILL(128) START STRICT NOSTRICT_INDUCTION
TARGET(LE, OSV2R10) NOTEST TUNE(3) NOUPCONV NOXPLINK COMPILED_ON_MVS

4    (   L) Function Name: CSTUFF#C
     (  WL) External Name: this_int_is_in_writable_static_and_its_name_wi
                           ll_warp_because_it_is_too_long
     (   L) Function Name: foo
     (  WL) External Name: CSTUFF#T
     (  WL) External Name: CSTUFF#S

*-------------------------------------------------------------------------*
*  Member Name: CPPIPANO                        (P) 2001/01/03 15:10:51 *
*                                                   5647A01    V2 R10   *
*-------------------------------------------------------------------------*

   User Comment:
AGGRCOPY(NOOVERLAP) ANSIALIAS ARCH(2) ARGPARSE NOCOMPACT NOCOMPRESS
NOCONVLIT NOCSECT CVFT DLL(NOCALLBACKANY) EXECOPS NOEXPORTALL
FLOAT(HEX, FOLD, NOAFP) NOGOFF NOGONUMBER NOIGNERRNO NOINITAUTO
IPA(NOLINK, NOOBJECT, COMPRESS, OPTIMIZE, NOGONUM) LANGLVL(EXTENDED)
NOLIBANSI NOLOCALE LONGNAME MAXMEM(2097152) NOOPTIMIZE PLIST(HOST)
REDIR NOROCONST ROSTRING ROUND(Z) NOSERVICE SPILL(128) START STRICT
NOSTRICT_INDUCTION TARGET(LE, OSV2R10) NOTEST(HOOK) TUNE(3) NOXPLINK
COMPILED_ON_MVS

    (I  L) Function Name: testeh()
    (I  L) Function Name: f1()
    (I  L) Function Name: a()
    (I  L) Function Name: A::areallyreallyreallyreallyreallyreallyreally
                          longnamefunction()
    (I  L) Function Name: A::operator+=(int)
    (I  L) Function Name: A::x()
    (I  L) External Name: i1
    (I  L) External Name: i2

*-------------------------------------------------------------------------*
*  Member Name:  CPPIPAO                        (P) 2001/01/03 15:10:51 *
*                                                   5647A01    V2 R10   *
*-------------------------------------------------------------------------*

   User Comment:
of IPA OBJECT AGGRCOPY(NOOVERLAP) ANSIALIAS ARCH(2) ARGPARSE NOCOMPACT
NOCOMPRESS NOCONVLIT NOCSECT CVFT DLL(NOCALLBACKANY) EXECOPS
NOEXPORTALL FLOAT(HEX, FOLD, NOAFP) NOGOFF NOGONUMBER NOIGNERRNO
NOINITAUTO IPA(NOLINK, OBJECT, COMPRESS, OPTIMIZE, NOGONUM)
LANGLVL(EXTENDED) NOLIBANSI NOLOCALE LONGNAME MAXMEM(2097152)
NOOPTIMIZE PLIST(HOST) REDIR NOROCONST ROSTRING ROUND(Z) NOSERVICE
SPILL(128) START STRICT NOSTRICT_INDUCTION TARGET(LE, OSV2R10)
NOTEST(HOOK) TUNE(3) NOXPLINK COMPILED_ON_MVS of OBJECT

    (   L) Function Name: testeh()
    (   L) Function Name: f1()
    (   L) Function Name: a()
    (   L) Function Name: A::areallyreallyreallyreallyreallyreallyreally
                          longnamefunction()
    (   L) Function Name: A::operator+=(int)
    (   L) Function Name: A::x()
    (  WL) External Name: i1
    (  WL) External Name: i2

*-------------------------------------------------------------------------*
*  Member Name: CPPIPAOB                        (P) 2001/01/03 15:10:51 *
*                                                   5647A01    V2 R10   *
*-------------------------------------------------------------------------*
```

```
   User Comment:
AGGRCOPY(NOOVERLAP) ANSIALIAS ARCH(2) ARGPARSE NOCOMPACT NOCOMPRESS
NOCONVLIT NOCSECT CVFT DLL(NOCALLBACKANY) EXECOPS NOEXPORTALL
FLOAT(HEX, FOLD, NOAFP) NOGOFF NOGONUMBER NOIGNERRNO NOINITAUTO
IPA(NOLINK, OBJONLY, COMPRESS, OPTIMIZE, NOGONUM) LANGLVL(EXTENDED)
NOLIBANSI NOLOCALE LONGNAME MAXMEM(2097152) NOOPTIMIZE PLIST(HOST)
REDIR NOROCONST ROSTRING ROUND(Z) NOSERVICE SPILL(128) START STRICT
NOSTRICT_INDUCTION TARGET(LE, OSV2R10) NOTEST(HOOK) TUNE(3) NOXPLINK
COMPILED_ON_MVS of OBJECT

    (   L) Function Name: testeh()
    (   L) Function Name: f1()
    (   L) Function Name: a()
    (   L) Function Name: A::areallyreallyreallyreallyreallyreallyreally
                          longnamefunction()
    (   L) Function Name: A::operator+=(int)
    (   L) Function Name: A::x()
    (  WL) External Name: i1
    (  WL) External Name: i2

*-------------------------------------------------------------------------*
*  Member Name: CPPXPLNK                       (P) 2001/00/00 13:51:25 *
*                                                  5647A01    V2 R10   *
*-------------------------------------------------------------------------*

   User Comment:
AGGRCOPY(NOOVERLAP) ANSIALIAS ARCH(2) ARGPARSE NOCOMPACT NOCOMPRESS
NOCONVLIT CSECT(CODE, CPPSTUFF#C) CSECT(STATIC, CPPSTUFF#S)
CSECT(TEST, CPPSTUFF#T) CVFT DLL(NOCALLBACKANY) EXECOPS NOEXPORTALL
FLOAT(HEX, FOLD, NOAFP) GOFF NOGONUMBER NOIGNERRNO NOINITAUTO NOIPA
LANGLVL(EXTENDED) NOLIBANSI NOLOCALE LONGNAME MAXMEM(2097152)
NOOPTIMIZE PLIST(HOST) REDIR NOROCONST ROSTRING ROUND(Z) NOSERVICE
SPILL(128) START STRICT NOSTRICT_INDUCTION TARGET(LE, OSV2R10)
NOTEST(HOOK) TUNE(3) XPLINK COMPILED_ON_MVS


    ( X L) Function Name: testeh()
    ( X L) Function Name: f1()
    ( X L) Function Name: a()
    ( X L) Function Name: A::areallyreallyreallyreallyreallyreallyreally
                          longnamefunction()
    ( X L) Function Name: A::operator+=(int)
    ( X L) Function Name: A::x()
    (  WL) External Name: i1
    (  WL) External Name: i2
    ( X L) Function Name: CPPSTUFF#C
    (  WL) External Name: CPPSTUFF#T
    (  WL) External Name: CPPSTUFF#S

*-------------------------------------------------------------------------*
*  Member Name:    CXOBJ                       (P) 2001/01/03 15:10:51 *
*                                                  5647A01    V2 R10   *
*-------------------------------------------------------------------------*

   User Comment:
AGGRCOPY(NOOVERLAP) NOALIAS ANSIALIAS ARCH(2) ARGPARSE NOCOMPACT
NOCOMPRESS NOCONVLIT NOCSECT NODLL(NOCALLBACKANY) EXECOPS NOEXPORTALL
FLOAT(HEX, FOLD, NOAFP) NOGOFF NOGONUMBER NOHWOPTS NOIGNERRNO
NOINITAUTO NOINLINE NOIPA LANGLVL(*EXTENDED) NOLIBANSI NOLOCALE
LONGNAME MAXMEM(2097152) OPTIMIZE(0) PLIST(HOST) REDIR RENT NOROCONST
NOROSTRING NOSERVICE SPILL(128) START STRICT NOSTRICT_INDUCTION
TARGET(LE, OSV2R10) NOTEST TUNE(3) NOUPCONV NOXPLINK COMPILED_ON_MVS

    (  WL) External Name: this_int_is_in_writable_static_and_its_name_wi
                          ll_warp_because_it_is_too_long
    (   L) Function Name: foo
```

```
================================================================================
|                        Symbol Definition Map                                 |
================================================================================


*------------------------------------------------------------------------------*
|  Symbol name: CSTUFF#C                                                       |
*------------------------------------------------------------------------------*

   From member:    CGOFF Type: Function (   L)

*------------------------------------------------------------------------------*
|  Symbol name: this_int_is_in_writable_static_and_its_name_will_warp_ |
|               because_it_is_too_long                                         |
*------------------------------------------------------------------------------*

  From member:    CGOFF Type: External (  WL)
  From member:    CXOBJ Type: External (  WL)

*------------------------------------------------------------------------------*
|  Symbol name: foo                                                            |
*------------------------------------------------------------------------------*

  From member:    CGOFF Type: Function (   L)
  From member:    CXOBJ Type: Function (   L)


*------------------------------------------------------------------------------*
|  Symbol name: CSTUFF#T                                                       |
*------------------------------------------------------------------------------*

  From member:    CGOFF Type: External (  WL)

*------------------------------------------------------------------------------*
|  Symbol name: CSTUFF#S                                                       |
*------------------------------------------------------------------------------*

  From member:    CGOFF Type: External (  WL)

*------------------------------------------------------------------------------*
|  Symbol name: testeh()                                                       |
*------------------------------------------------------------------------------*

  From member:  CPPIPAO Type: Function (   L)
  From member: CPPIPAOB Type: Function (   L)
  From member: CPPXPLNK Type: Function ( X L)
  From member: CPPIPANO Type: Function (I  L)

*------------------------------------------------------------------------------*
|  Symbol name: f1()                                                           |
*------------------------------------------------------------------------------*

  From member:  CPPIPAO Type: Function (   L)
  From member: CPPIPAOB Type: Function (   L)
  From member: CPPXPLNK Type: Function ( X L)
  From member: CPPIPANO Type: Function (I  L)

*------------------------------------------------------------------------------*
|  Symbol name: a()                                                            |
*------------------------------------------------------------------------------*

  From member:  CPPIPAO Type: Function (   L)
  From member: CPPIPAOB Type: Function (   L)
  From member: CPPXPLNK Type: Function ( X L)
  From member: CPPIPANO Type: Function (I  L)
```

```
*-------------------------------------------------------------------------*
|  Symbol name: A::areallyreallyreallyreallyreallyreallyreallylongname    |
|               function()                                                |
*-------------------------------------------------------------------------*

  From member:  CPPIPAO Type: Function (   L)
  From member: CPPIPAOB Type: Function (   L)
  From member: CPPXPLNK Type: Function ( X L)
  From member: CPPIPANO Type: Function (I  L)

*-------------------------------------------------------------------------*
|  Symbol name: A::operator+=(int)                                        |
*-------------------------------------------------------------------------*

  From member:  CPPIPAO Type: Function (   L)
  From member: CPPIPAOB Type: Function (   L)
  From member: CPPXPLNK Type: Function ( X L)
  From member: CPPIPANO Type: Function (I  L)



*-------------------------------------------------------------------------*
|  Symbol name: A::x()                                                    |
*-------------------------------------------------------------------------*

  From member:  CPPIPAO Type: Function (   L)
  From member: CPPIPAOB Type: Function (   L)
  From member: CPPXPLNK Type: Function ( X L)
  From member: CPPIPANO Type: Function (I  L)

*-------------------------------------------------------------------------*
|  Symbol name: i1                                                        |
*-------------------------------------------------------------------------*

  From member:  CPPIPAO Type: External (  WL)
  From member: CPPIPAOB Type: External (  WL)
  From member: CPPXPLNK Type: External (  WL)
  From member: CPPIPANO Type: External (I  L)

*-------------------------------------------------------------------------*
|  Symbol name: i2                                                        |
*-------------------------------------------------------------------------*

  From member:  CPPIPAO Type: External (  WL)
  From member: CPPIPAOB Type: External (  WL)
  From member: CPPXPLNK Type: External (  WL)
  From member: CPPIPANO Type: External (I  L)

*-------------------------------------------------------------------------*
|  Symbol name: CPPSTUFF#C                                                |
*-------------------------------------------------------------------------*
  From member: CPPXPLNK Type: Function ( X L)

*-------------------------------------------------------------------------*
|  Symbol name: CPPSTUFF#T                                                |
*-------------------------------------------------------------------------*

  From member: CPPXPLNK Type: External (  WL)

*-------------------------------------------------------------------------*
|  Symbol name: CPPSTUFF#S                                                |
*-------------------------------------------------------------------------*

  From member: CPPXPLNK Type: External (  WL)

========  E N D   O F   O B J E C T   L I B R A R Y   M A P  ==========
```

1 Map Heading

The heading contains the product number, the library version and release number, and the date and the time the Object Library Utility step began. The name of the library immediately follows the heading. To the right of the library name is the start time of the last Object Library Utility step that updated the Object Library Utility-directory.

2 Member Heading

The product number of the processor that produced the object file follows the name of the object file member. If the `END` record in the object file does not have the processor information in the appropriate format, the `Processor ID` field does not appear.

The `Timestamp` field appears in `yyyy/mm/dd` format. A letter that is enclosed in parentheses indicates the meaning of the timestamp. That is, the Object Library Utility retains a timestamp for each member and selects the time according to the following hierarchy:

(P) indicates that the timestamp is extracted from the object file from the date form or the timestamp form of `#pragma comment`, whichever comes first.

(D) indicates that the timestamp is based on the time that the Object Library Utility `DIR` command was last issued.

(T) indicates that the timestamp is the time that the `ADD` command was issued for the member.

3 User Comments

Displays the user form of comments that `#pragma comment` generated. These comments are extracted from the `END` record. You can add such comments on multiple `END` records and have them displayed in the listing. See the *C/C++ Language Reference* more information on the `END` record.

4 Symbol Information

Immediately following `Member Heading` and user comments is a list of the defined objects that the member contains. Each symbol is prefixed by `Type` information that is enclosed in parentheses and either `External Name` or `Function Name`. `Function Name` will appear, provided the object file was compiled with the `LONGNAME` option and the symbol is the name of a defined external function. In all other cases, `External Name` is displayed. The Type field gives additional information on each symbol. That is:

'L' indicates that the name is a long name. An long name is an external C++ name in an object file or an external non-C++ name in an object file produced by compiling with the `LONGNAME` option.

'S' indicates that the name is a short name. A short name is an external non-C++ name in an object file produced by compiling with the `NOLONGNAME` option. Such a name is up to 8 characters long and single case.

'W' indicates that this is a writable static object. If it is not present, then this is not a writable static object.

'X' indicates that the name is compiled XPLINK.

Note: `WL` indicates that the symbol is both a long name and in writable static.

Filter Utility

This chapter describes how to use the CXXFILT utility to convert mangled names to demangled names.

When C/C++ for z/VM compiles a C++ program, it has the ability to encode function names. It also has the ability to encode other identifiers to include type and scoping information. This encoding process is called mangling. Mangled names ensure type-safe linking.

Use the CXXFILT utility to convert these mangled names to demangled names. The utility copies the characters from either a given file or from standard input, to standard output. It replaces all mangled names with their corresponding demangled names.

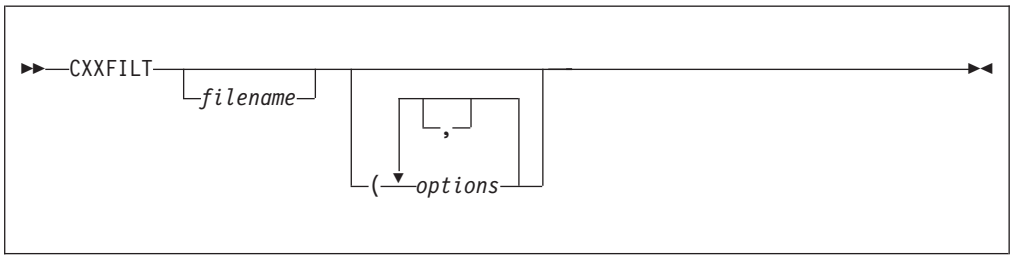
The CXXFILT utility demangles any of the following classes of mangled names when the appropriate options are specified.

regular names
Names that appear within the context of a function name or a member variable. For example, the mangled name __ls__7ostreamFPCc is demangled as ostream::operator<<(const char*).

class names
Includes stand-alone class names that do not appear within the context of a function name or a member variable. For example, the stand-alone class name Q2_1X1Y is demangled as X::Y.

special names
Special compiler-generated class objects. For example, the compiler-generated symbol name __vft1X is demangled as X::virtual-fn-table-ptr.

The CXXFILT utility is run under VM/CMS by using the CXXFILT EXEC. The format of the parameters for the CXXFILT EXEC is:



where filename is the file that contain the mangled names to be demangled. If you specify no file name, CXXFILT reads from stdin.

CXXFILT Options

SYMMAP|NOSYMMAP
DEFAULT: NOSYMMAP

Produces a symbol map on standard output. This map contains a list of the mangled names and their corresponding demangled names. The map only displays the first 40 bytes of each demangled name; it truncates the rest. Mangled names are not truncated.

If an input mangled name does not have a demangled version, the symbol mapping does not display it.

The symbol mapping is displayed after the end of the input stream is encountered, and after CXXFILT terminates.

SIDEBYSIDE|NOSIDEBYSIDE
DEFAULT: NOSIDEBYSIDE

Each mangled name that is encountered in the input stream is displayed beside its corresponding demangled name. If you do not specify this option, then only the demangled names are printed. In either case, trailing characters in the input name that are not part of a mangled name appear next to the demangled name. For example, if an extraneous xxxx is input with the mangled name pr__3FOOF, then the SIDEBYSIDE option would produce this result:

```
FOO::pr()       pr__3FOOFvxxxx
```

WIDTH(width)|NOWIDTH
DEFAULT: NOWIDTH

Prints demangled names in fields, width characters wide. If the name is shorter than width, it is padded on the right with blanks; if longer, it is truncated to width. The value of width must be greater than 0. If width is greater than the record width, then the output is wrapped.

REGULARNAME|NOREGULARNAME
DEFAULT: REGULARNAME

This option demangles regular names such as pr__3FOOFv.

The mangled name that is supplied to CXXFILT is treated as a regular name by default. Specifying the NOREGULARNAME option will turn the default off. For example, specifying the CLASSNAME option without the NOREGULARNAME option will cause CXXFILT to treat the mangled name as either a regular name or stand-alone class name.

CLASSNAME|NOCLASSNAME
DEFAULT: NOCLASSNAME

This option demangles stand-alone class names such as Q2_1X1Y.

To request that the mangled names be treated as stand-alone class names only, and never as a regular name, use both CLASSNAME and NOREGULARNAME.

SPECIALNAME|NOSPECIALNAME
DEFAULT: NOSPECIALNAME

Demangles special names, such as compiler-generated symbol names, for example __vft1X.

To request that the mangled names be treated as special names only, and never as regular names, use CXXFILT (SPECIALNAME NOREGULARNAME.

Unknown Type of Name
If you cannot specify the type of name, use CXXFILT (SPECIALNAME CLASSNAME. This causes CXXFILT to attempt to demangle the name in the following order:

1. Regular name
2. Stand-alone class name

3. Special name

Running CXXFILT under VM/CMS

The CXXFILT EXEC accepts input by two methods: from stdin or from a file.

With the first method, enter names after invoking CXXFILT. You can specify one or more names on one or more lines. The output is displayed after you press Enter. Names that are successfully demangled, as well as those which are not demangled, are displayed in the same order as they were entered. To indicate end of input, enter /*.

In the following example, CXXFILT treats mangled names as regular names, produces a symbol mapping, and uses a field width 32 characters wide.

```
user>  CXXFILT (SYMMAP WIDTH(32)
user>  pr__3FOOFvxxxx
reply< FOO::pr()                        xxxx
user>  __ls__7ostreamFPCc
reply> ostream::operator<<(const char*)
user>  __vft1X
reply> X::virtual-fn-table-ptr
user>  /*

reply> C++ Symbol Mapping
reply> demangled                                mangled
reply> ---------                                -------
reply>
reply> FOO::pr()                                pr__3FOOFv
reply> ostream::operator<<(const char*)         __ls__7ostreamFPCc
reply> X::virtual-fn-table-ptr                  __vft1X
```

Notes:

1. Because the trailing characters xxxx in the input name pr__3FOOFvxxxx are not part of a valid mangled name, and the SIDEBYSIDE option is not on, the trailing characters are not demangled.

   In the symbol mappings, the trailing characters xxxx are not displayed.
2. The symbol mapping is displayed only after /* requests CXXFILT termination.

The second method of giving input to CXXFILT is to supply it in a file. CXXFILT supports fixed and variable file record formats. Each line of the file can have one or more names separated by space. In the example below, mangled names are treated either as regular names or as special names (the special names are compiler-generated symbol names). Demangled names are printed in fields 35 characters wide, and output is in side-by-side format.

NAMES FILE contains the following two mangled names:

```
pr__3FOOFv
__vft1X
```

Entering the following command:

```
CXXFILT NAMES FILE (SPECIALNAME WIDTH(35) SIDEBYSIDE
```

produces the following output:

```
 FOO::pr()                       pr__3FOOFv
 X::virtual-fn-table-ptr         __vft1X
```

CXXFILT terminates when it reads the end-of-file.

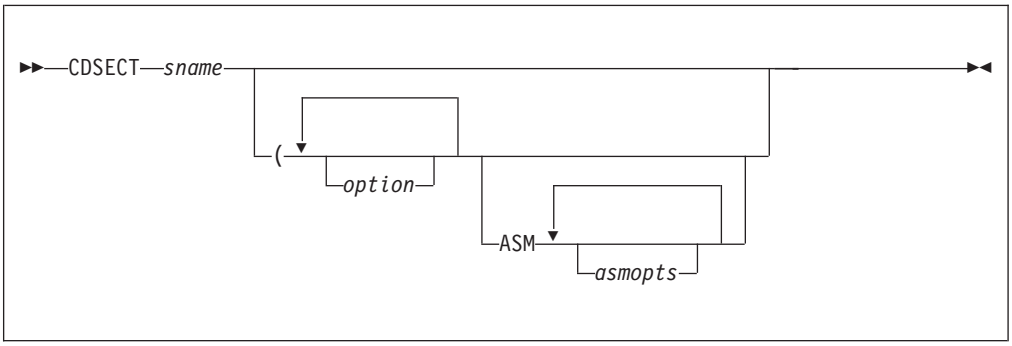
DSECT Conversion Utility

This chapter describes how to use the DSECT conversion utility.

The DSECT conversion utility generates a C structure to map an assembler DSECT. This utility is used when a C program calls or is called by an Assembler program and a C structure is required to map the area passed.

The source for the assembler DSECT is assembled using the High-Level Assembler specifying the ADATA option. (See *IBM High Level Assembler Programmer's Guide*, for a description of the ADATA option.) The DSECT utility then reads the SYSADATA file produced by the High Level Assembler and produces a file containing the C structure according to the options specified.

The DSECT utility is run under VM/CMS by using the CDSECT EXEC. The format of the parameters for the CDSECT EXEC is:



where sname specifies the file name of the assembler source program containing the required section, and options are any valid DSECT utility options. If you specify ASM, any following options must be High Level Assembler options. The ADATA option is specified by default.

```
KNOWN:   - The assembler source name is TESTASM ASSEMBLE A.
         - The required DSECT Utility options are EQU(BIT).

USE THE FOLLOWING COMMAND:
         CDSECT TESTASM ( EQU(BIT)
```

Figure 36. Running the DSECT Utility under CMS

When the CDSECT command is executed, the High Level Assembler is executed with the required options. The DSECT utility is then executed with the specified options. A report is produced in file "sname DMAP A1". The C structure produced is written to a file "sname STRUCT A1" unless the OUTPUT option is specified.

If the assembler source requires macros or copy members from a MACLIB, issue the GLOBAL MACLIB command to set up the required MACLIBs before issuing the CDSECT command.

DSECT Utility Options

The options that you can use to control the generation of the C structure are as follows. You can specify them in upper- or lowercase, separating them by spaces or commas.

Table 7. DSECT Utility Options, Abbreviations, and IBM-Supplied Defaults

DSECT Utility Option	Abbreviated Name	IBM Supplied Default
SECT[(*name*,...)]	None	SECT(ALL)
BITF0XL|NOBITF0XL	BITF|NOBITF	NOBITF0XL
COMMENT[(*delim*,...)]|NOCOMMENT	COM|NOCOM	COMMENT
DEFSUB|NODEFSUB	DEF|NODEF	DEFSUB
EQUATE[(*suboptions*,...)]|NOEQUATE	EQU|NOEQU	NOEQUATE
HDRSKIP[(*length*)]|NOHDRSKIP	HDR(*length*)|NOHDR	NOHDRSKIP
LOCALE(*name*)|NOLOCALE	LOC|NOLOC	NOLOCALE
INDENT[(*count*)]|NOINDENT	IN(*count*)|NOIN	INDENT(2)
LOWERCASE|NOLOWERCASE	LC|NOLC	LOWERCASE
OPTFILE(*filename*)|NOOPTFILE	OPTF|NOOPTF	NOOPTFILE
PPCOND[(*switch*)]|NOPPCOND	PP(*switch*)|NOPP	NOPPCOND
SEQUENCE|NOSEQUENCE	SEQ|NOSEQ	NOSEQUENCE
UNNAMED|NOUNNAMED	UNN|NOUNN	NOUNNAMED
OUTPUT[(*filename*)]	OUT[(*filename*)]	OUTPUT(DD:EDCDSECT)
RECFM[(*recfm*)]	None	C Library defaults
LRECL[(*lrecl*)]	None	C Library defaults
BLKSIZE[(*blksize*)]	None	C Library defaults

SECT
DEFAULT: SECT(ALL)

The SECT option specifies the section names for which C structures are to be produced. The section names can be either CSECT or DSECT names. They must exist in the SYSADATA file produced by the Assembler. If you do not specify the SECT option or if you specify SECT(ALL), C structures are produced for all CSECTs and DSECTs defined in the SYSADATA file, except for private code and unnamed DSECTs.

If the High Level Assembler is run with the BATCH option, only the section names defined within the first program can be specified on the SECT option. If you specify SECT(ALL) (or select it by default), only the sections from the first program are selected.

BITF0XL | NOBITF0XL
DEFAULT: NOBITF0XL

Specify the BITF0XL option when the bit fields are mapped into a flag byte as in the following example:

```
FLAGFLD   DS   F
          ORG  FLAGFLD+0
B1FLG1    DC   0XL(B'10000000')'00'    Definition for bit 0 of 1st byte
B1FLG2    DC   0XL(B'01000000')'00'    Definition for bit 1 of 1st byte
B1FLG3    DC   0XL(B'00100000')'00'    Definition for bit 2 of 1st byte
B1FLG4    DC   0XL(B'00010000')'00'    Definition for bit 3 of 1st byte
B1FLG5    DC   0XL(B'00001000')'00'    Definition for bit 4 of 1st byte
B1FLG6    DC   0XL(B'00000100')'00'    Definition for bit 5 of 1st byte
B1FLG7    DC   0XL(B'00000010')'00'    Definition for bit 6 of 1st byte
B1FLG8    DC   0XL(B'00000001')'00'    Definition for bit 7 of 1st byte
          ORG  FLAGFLD+1
```

```
B2FLG1    DC   0XL(B'10000000')'00'    Definition for bit 0 of 2nd byte
B2FLG2    DC   0XL(B'01000000')'00'    Definition for bit 1 of 2nd byte
B2FLG3    DC   0XL(B'00100000')'00'    Definition for bit 2 of 2nd byte
B2FLG4    DC   0XL(B'00010000')'00'    Definition for bit 3 of 2nd byte
```

When the bit fields are mapped as shown in the above example, the bit fields can be tested using the following code:

```
TM   FLAGFLD,L'B1FLG          Test bit 0 of byte 1
Bx   label                    Branch if set/not set
```

When you specify the BITF0XL option, the length attribute of the following fields is used to provide the mapping for the bits within the flag bytes.

The length attribute of the following fields is used to map the bit fields if a field conforms to the following rules:

- Does not have a duplication factor of zero.
- Has a length between 1 and 4 bytes and does not have a bit length.
- Does not have more than 1 nominal value.

and the following fields conform to the following rules:

- Has a Type attribute of 'B', 'C' or 'X'.
- Has the same offset as the field (or consecutive fields have overlapping offsets).
- Has a duplication factor of zero.
- Does not have more than 1 nominal value.
- Has a length attribute between 1 and 255 and does not have a bit length.
- The length attribute maps one bit or consecutive bits. For example, B'10000000' or B'11000000', but not B'10100000'.

The fields must be on consecutive lines and must overlap a named field. If the fields above are used to define the bits for a field, any EQU statements following the field are not used to define the bit fields.

The following fields are used to define the bit fields as long as they map consecutive bits. If two consecutive fields are equivalent, the second field is skipped.

COMMENT | NOCOMMENT

DEFAULT: COMMENT

The COMMENT option specifies whether the comments on the line where the field is defined will be placed in the C structure produced.

If you specify the COMMENT option without a delimiter, the entire comment is placed in the C structure.

If you specify a delimiter, any comments following the delimiter are skipped and are not placed in the C structure. You can remove changes that are flagged with a particular delimiter. The delimiter cannot contain imbedded spaces or commas. The case of the delimiter and comment text is not significant. You can specify up to 10 delimiters, and they can contain up to 10 characters each.

DEFSUB | NODEFSUB

DEFAULT: DEFSUB

The DEFSUB option specifies whether #define directives will be built for fields that are part of a union or substructure.

If the DEFSUB option is in effect, fields within a substructure or union have the field names prefixed by an underscore. A #define directive is written at the end of the structure to allow the field name to be specified directly as in the following example:

```
_Packed struct dsect_name {
  int           field1;
  _Packed struct {
    int           _subfld1;
    short int     _subfld2;
    unsigned char _subfld3[4];
    } field2;
  }
#define subfld1  field2._subfld1
#define subfld2  field2._subfld2
#define subfld3  field2._subfld3
```

If the DEFSUB option is in effect, the fields prefixed by an underscore may match the name of another field within the structure. No warning is issued.

EQUATE | NOEQUATE

DEFAULT: NOEQUATE

The EQUATE option specifies whether the EQU statements following a field are to be used to define bit fields, to generate #define directives, or are to be ignored.

The suboptions specify how the EQU statement is used. You can specify one or more of the suboptions, separating them by spaces or commas. If you specify more than one suboption, the EQU statements following a field are checked to see if they are valid for the first suboption. If so, they are formatted according to that option. Otherwise, the subsequent suboptions are checked to see if they are applicable.

If you specify the EQUATE option without suboptions, EQUATE(BIT) is used. If you specify NOEQUATE (or select it by default), the EQU statements following a field are ignored.

You can specify the following suboptions for the EQUATE option:

BIT indicates that the value for an EQU statement is used to define the bits for a field where the field conforms to the following rules:

- Does not have a duplication factor of zero.
- Has a length between 1 and 4 bytes and has a bit length that is a multiple of 8.
- Does not have more than 1 nominal value.

and the EQU statements following the field conform to the following rules:

- The value for the EQU statements following the field mask consecutive bits (for example, X'80' followed by X'40').
- The value for an EQU statement masks one bit or consecutive bits for example, B'10000000' or B'11000000', but not B'10100000'.
- Where the length of the field is greater than 1 byte, the bits for the remaining bytes can be defined by providing the EQU statements for the second byte after the EQU statement for the first byte.
- The value for the EQU statement is not a relocatable value.

When you specify EQUATE(BIT), the EQU statements are converted as in the following example:

```
FLAGFLD  DS   H
FLAG21   EQU  X'80'
FLAG22   EQU  X'40'
FLAG23   EQU  X'20'
FLAG24   EQU  X'10'
FLAG25   EQU  X'08'
FLAG26   EQU  X'04'
FLAG27   EQU  X'02'
FLAG28   EQU  X'01'
FLAG2A   EQU  X'80'
FLAG2B   EQU  X'40'
_Packed struct dsect_name {
  unsigned int flag21   : 1,
               flag22   : 1,
               flag23   : 1,
               flag24   : 1,
               flag25   : 1,
               flag26   : 1,
               flag27   : 1,
               flag28   : 1,
               flag2a   : 1,
               flag2b   : 1,
                        : 6;
  }
```

BITL indicates that the length attribute for an EQU statement is used to define the bits for a field where the field conforms to the following rules:

- Does not have a duplication factor of zero.
- Has a length between 1 and 4 bytes and has a bit length that is a multiple of 8.
- Does not have more than 1 nominal value.

and the EQU statements following the field conform to the following rules:

- The value specified for the EQU statement has the same or overlapping offset as the field.
- The length attribute for the EQU statement is between 1 and 255.
- The length attribute for the EQU statement masks one bit or consecutive bits, for example, B'10000000' or B'11000000', but not B'10100000'.
- The value for the EQU statement is a relocatable value.

When you specify EQUATE(BITL), the EQU statements are converted as in the following example:

```
BYTEFLD   DS   F
B1FLG1    EQU  BYTEFLD+0,B'10000000'
B1FLG2    EQU  BYTEFLD+0,B'01000000'
B1FLG3    EQU  BYTEFLD+0,B'00100000'
B1FLG4    EQU  BYTEFLD+0,B'00010000'
B1FLG5    EQU  BYTEFLD+0,B'00001000'
B1FLG6    EQU  BYTEFLD+0,B'00000100'
B1FLG7    EQU  BYTEFLD+0,B'00000010'
B1FLG8    EQU  BYTEFLD+0,B'00000001'
B2FLG1    EQU  BYTEFLD+1,B'10000000'
B2FLG2    EQU  BYTEFLD+1,B'01000000'
B2FLG3    EQU  BYTEFLD+1,B'00100000'
B2FLG4    EQU  BYTEFLD+1,B'00010000'
_Packed struct dsect_name {
  unsigned int b1flg1  : 1,
               b1flg2  : 1,
               b1flg3  : 1,
               b1flg4  : 1,
```

```
                    b1flg5  : 1,
                    b1flg6  : 1,
                    b1flg7  : 1,
                    b1flg8  : 1,
                    b2flg1  : 1,
                    b2flg2  : 1,
                    b2flg3  : 1,
                    b2flg4  : 1,
                            : 20;
  }
```

DEF indicates that the EQU statements following a field are used to build #define directives to define the possible values for a field. The #define directives are placed after the end of the C structure. The EQU statements should not specify a relocatable value.

When you specify EQUATE(DEF), the EQU statements are converted as in the following example:

```
FLAGBYTE DS   X
FLAG1    EQU  X'80'
FLAG2    EQU  X'20'
FLAG3    EQU  X'10'
FLAG4    EQU  X'08'
FLAG5    EQU  X'06'
FLAG6    EQU  X'01'
_Packed struct dsect_name {
  unsigned char flagbyte;
  }
/* Values for flagbyte field */
#define flag1 0x80
#define flag2 0x20
#define flag3 0x10
#define flag4 0x08
#define flag5 0x06
#define flag6 0x01
```

HDRSKIP | NOHDRSKIP

DEFAULT: NOHDRSKIP

The HDRSKIP option specifies that the fields within the specified number of bytes from the start of the section are to be skipped. Use this option where a section has a header that is not required in the C structure produced.

The value specified on the HDRSKIP option indicates the number of bytes at the start of the section that are to be skipped. HDRSKIP(0) is equivalent to NOHDRSKIP.

In the following example, if you specify HDRSKIP(8), the first two fields are skipped and only the remaining two fields are built into the structure.

```
SECTNAME DSECT
PREFIX1  DS   CL4
PREFIX2  DS   CL4
FIELD1   DS   CL4
FIELD2   DS   CL4
_Packed struct sectname {
  unsigned char field1[4];
  unsigned char field2[4];
  }
```

If the value specified for the HDRSKIP option is greater than the length of the section, the C structure is not be produced for that section.

INDENT | NOINDENT

DEFAULT: INDENT(2)

The INDENT option specifies the number of character positions that the fields, unions, and substructures are indented. Turn off indentation by specifying INDENT(0) or NOINDENT. The maximum value that you can specify for the INDENT option is 32767.

LOCALE | NOLOCALE

The LOCALE(*name*) specifies the name of a locale to be passed to the setlocale() function. Specifying LOCALE without the *name* parameter is equivalent to passing the NULL string to the setlocale() function.

The structure produced contains the left and right brace, and left and right square bracket, backslash, and number sign which have different code point values for the different code pages. When the LOCALE option is specified, and these characters are written to the output file, the code point from the LC_SYNTAX category for the specified locale is used.

The default is NOLOCALE.

You can abbreviate the option to LOC(*name*) or NOLOC.

LOWERCASE | NOLOWERCASE

DEFAULT: LOWERCASE

The LOWERCASE option specifies whether the field names within the C structure are to be converted to lowercase or left as entered. If you specify LOWERCASE, all the field names are converted to lowercase. If you specify NOLOWERCASE, the field names are built into the structure in the case in which they were entered in the assembler section.

OPTFILE | NOOPTFILE

The OPTFILE(*filename*) option specifies the filename containing the records that specify the options to be used for processing the sections. The records must be as follows:

- The lines must begin with the SECT option, with only one section name specified. The options following determine how the structure is produced for the specified section. The section name must only be specified once.
- The lines may contain the options BITF0XL, COMMENT, DEFSUB, EQUATE, HDRSKIP, INDENT, LOWERCASE, PPCOND, and UNNAMED, separated by spaces or commas. These override the options specified on the command line for the section.

The OPTFILE option is ignored if the SECT option is also specified on the command line.

The default is NOOPTFILE.

You can abbreviate the option to OPTF(*filename*) or NOOPTF.

PPCOND | NOPPCOND

DEFAULT: NOPPCOND

The PPCOND option specifies whether preprocessor directives will be built around the structure definition to prevent duplicate definitions.

If you specify PPCOND, the following are built around the structure definition.

```
#ifndef switch
#define switch
  .
```

```
   .
   .
   structure definition for section
   .
   .
   .
#endif
```

where *switch* is the switch specified on the PPCOND option or the section name prefixed and suffixed by two underscores, for example, __name__.

If you specify a switch, the #ifndef and #endif directives are placed around all structures that are produced. If you do not specify a switch, the #ifndef and #endif directives are placed around each structure produced.

SEQUENCE | NOSEQUENCE

DEFAULT: NOSEQUENCE

The SEQUENCE option specifies whether sequence numbers will be placed in columns 73 to 80 of the output record. If you specify the SEQUENCE option, the C structure is built into columns 1 to 72 of the output record and sequence numbers are placed in columns 73 to 80. If you specify NOSEQUENCE (or select it by default), sequence numbers are not generated and the C structure is built within all available columns in the output record.

If the record length for the output file is less than 80 characters, the SEQUENCE option is ignored.

UNNAMED | NOUNNAMED

DEFAULT: NOUNNAMED

The UNNAMED option specifies that names are not generated for the unions and substructures within the main structure.

OUTPUT

DEFAULT: OUTPUT(DD:EDCDSECT)

The C structures produced are, by default, written to the EDCDSECT DD statement. You can use the OUTPUT option to specify an alternative DD statement or data-set name to write the C structure. You can specify any valid file name up to 60 characters in length. The file name specified will be passed to fopen() as entered.

RECFM

DEFAULT: C Library default

The RECFM option specifies the record format for the file to be produced. You can specify up to 10 characters. If it is not specified, the C library defaults are used.

LRECL

DEFAULT: C Library default

The LRECL option specifies the logical record length for the file to be produced. The logical record length specified must not be greater than 32767. If it is not specified, the C library defaults will be used.

BLKSIZE

DEFAULT: C Library default

The BLKSIZE option specifies the block size for the file to be produced. The block size specified must not be greater than 32767. If it is not specified, the C library defaults will be used.

Generation of C Structures

The C structure is produced as follows according to the options in effect.

- The section name is used as the structure name. The structure is generated with the _Packed attribute to ensure it matches the assembler section.

  Whenever you specify the structure name, you should also specify the _Packed attribute.
- Any nonalphanumeric characters in the section or field names are converted to underscores. Duplicate names may be generated when the field names are identical except for the national character. No warning is issued.
- Where fields overlap, a substructure or union is built within the main structure. A substructure is produced where possible. When substructures and unions are built, the structure and unions names are generated by the DSECT utility.
- The substructures and unions within the main structure are indented according to the INDENT option unless the record length is too small to permit any further indentation.
- Fillers are added within the structure when required. A filler name is generated by the DSECT utility.
- Where there is no direct equivalent for an assembler definition within the C language, the field is defined as a character field.
- If a field has a duplication factor of zero, but cannot be used as a structure name, the field is defined as though the duplication factor of zero was eliminated.
- Where a line within the assembler input consists of an operand with a duplication factor of zero (for alignment), followed by the field definition, the first operand is skipped. For example:

  ```
  FIELDA    DS  0F,CLB
  ```

  is treated as though the following was specified:

  ```
  FIELDA    DS  CLB
  ```
- When the COMMENT option is in effect, the comment on the line following the definition of the field is placed in the C structure. The comment is placed on the same line as the field definition where possible, or on the following line.

  /* is removed from the beginning of comments and */ is removed from the end of comments. Any remaining instances of */ in the comment are converted to **.

Each field within the section is converted to a field within the C structure as shown in the following examples:

- Bit length fields

  If the field has a bit length that is not a multiple of 8, it is converted as follows. Otherwise, it is converted according to the field type.

DS CL.n	`unsigned int name : n;` where n is from 1 to 31.
DS CL.n	`unsigned char name[x];` where n is greater than 32. x will be the number of bytes required (that is, the bit length / 8 + 1).
DS 5CL.n	unsigned char name[x]; where x will be the number of bytes required (that is, the duplication factor * bit length / 8 + 1).
- Characters

DS C `unsigned char name;`
DS CL2 `unsigned char name[2];`
DS 4CL2 `unsigned char name[4][2];`

- Graphic Characters

  DS G `wchar_t name;`
  DS GL1 `unsigned char name;`
  DS GL2 `wchar_t name;`
  DS GL3 `unsigned char name[3];`
  DS 4GL1 `unsigned char name[4];`
  DS 4GL2 `wchar_t name[4];`
  DS 4GL3 `unsigned char name[4][3];`

- Hexadecimal Characters

  DS X `unsigned char name;`
  DS XL2 `unsigned char name[2];`
  DS 4XL2 `unsigned char name[4][2];`

- Binary fields

  DS B `unsigned char name;`
  DS BL2 `unsigned char name[2];`
  DS 4BL2 `unsigned char name[4][2];`

- Half and Fullword Fixed-point

  DS F `int name;`
  DS H `short int name;`
  DS FL1 or HL1 `char name;`
  DS FL2 or HL2 `short int name;`
  DS FL3 or HL3 `int name : 24;`
  DS FLn or HLn `unsigned char name[n];` where n is greater than 4.

  DS 4F `int name[4];`
  DS 4H `short int name[4];`
  DS 4FL1 or 4HL1 `char name[4];`
  DS 4FL2 or 4HL2 `short int name[4];`
  DS 4FL3 or 4HL3 `unsigned char name[4][3];`
  DS 4FLn or 4HLn `unsigned char name[4][n];` where n is greater than 4.

- Floating Point

  DS E `float name;`
  DS D `double name;`
  DS L `long double name;`
  DS 4E `float name[4];`
  DS 4D `double name[4];`
  DS 4L `long double name[4];`
  DS EL4 or DL4 or LL4 `float name;`
  DS EL8 or DL8 or LL8 `double name;`
  DS LL16 `long double name;`
  DS E, D or L `unsigned char name[n];` where n is other than 4, 8 or 16.

- Packed Decimal

  DS P `unsigned char name;`
  DS PL2 `unsigned char name[2];`
  DS 4PL2 `unsigned char name[4][2];`

- Zoned Decimal

  DS Z `unsigned char name;`
  DS ZL2 `unsigned char name[2];`
  DS 4ZL2 `unsigned char name[4][2];`

- Address

DS A	`void          *name;`
DS AL1	`unsigned char  name;`
DS AL2	`unsigned short name;`
DS AL3	`unsigned int   name : 24;`
DS 4A	`void          *name[4];`
DS 4AL1	`unsigned char  name[4];`
DS 4AL2	`unsigned short name[4];`
DS 4AL3	`unsigned char  name[4][3];`

- Y-type Address

DS Y	`unsigned short name;`
DS YL1	`unsigned char  name;`
DS 4Y	`unsigned short name[4];`
DS 4YL1	`unsigned char  name[4];`

- S-type Address (Base and displacement)

DS S	`unsigned short name;`
DS SL1	`unsigned char  name;`
DS 4S	`unsigned short name[4];`
DS 4SL1	`unsigned char  name[4];`

- External Symbol Address

DS V	`void          *name;`
DS VL3	`unsigned int   name : 24;`
DS 4V	`void          *name[4];`
DS 4VL3	`unsigned char  name[4][3];`

- External Dummy Section Offset

DS Q	`unsigned int   name;`
DS QL1	`unsigned char  name;`
DS QL2	`unsigned short name;`
DS QL3	`unsigned int   name : 24;`
DS 4Q	`unsigned int   name[4];`
DS 4QL1	`unsigned char  name[4];`
DS 4QL2	`unsigned short name[4];`
DS 4QL3	`unsigned char  name[4][3];`

- Channel Command Words

  When a CCW, CCW0, or CCW1 assembler instruction is present within the section, a typedef ccw0_t or ccw1_t is defined to map the format of the CCW.

  The CCW, CCW0 or CCW1 is built into the C structure as follows:

CCW cc,addr,flags,count	`ccw0_t         name;`
CCW0 cc,addr,flags,count	`ccw0_t         name;`
CCW1 cc,addr,flags,count	`ccw1_t         name;`

Code Set and Locale Utilities

This chapter describes the code set conversion utilities which help you convert a file from one code set to another and the `localedef` utility which allows you to define the language and cultural conventions used in your environment.

Code Set Conversion Utilities

The Code Set Conversion facilities that you may find useful prior to compiling are:

iconv Converts a file from one code set encoding to another. It can be used to convert C source code before compilation or to convert input files.

genxlt Generates a translate table for use by the `iconv` utility and iconv functions to perform code set conversion. It can be used to build code set

conversions for existing code pages that are not supplied with C, or to build code set conversions for existing code pages. The `iconv_open()`, `iconv()`, and `iconv_close()` functions are called from the `iconv` utility to perform code set translation. These functions can be called from any program requiring code set translation. For more information on these functions, refer to the *C/C++ for z/VM Run-Time Library Reference*.

iconv Utility

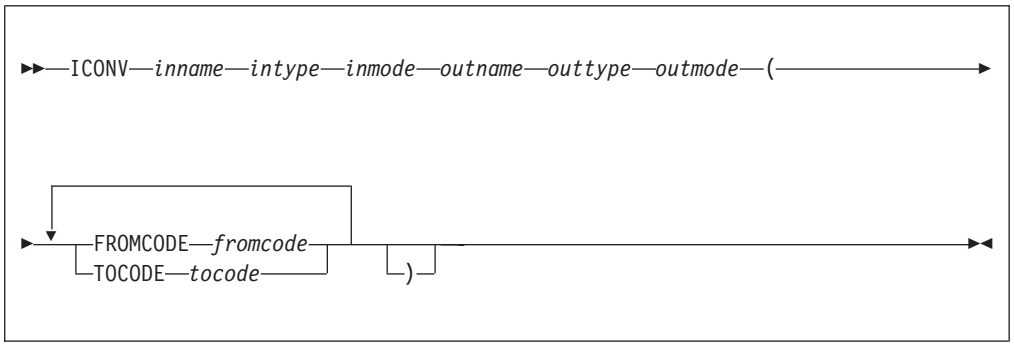
The `iconv` utility converts the characters from the input file from one coded character set (code set) definition to another code set definition, and writes the characters to the output file.

The `iconv` utility uses the `iconv_open()`, `iconv()`, and `iconv_close()` functions to convert the input file records from the coded character set definition for the input code page to the coded character set definition for the output code page. There is one record in the output file for each record in the input file. No padding or truncation of records is performed.

When conversions are performed between single-byte code pages, the output records are the same length as the input records. When conversions are performed between double-byte code pages, the output records may be longer or shorter than the input records because the shift-out and shift-in characters may be added or removed.

A REXX EXEC `ICONV` is provided to invoke the `iconv` utility to copy the input file to the output file and convert the characters from the input code page to the output code page.

The general parameters of the `ICONV` EXEC are as follows:



Where:

inname
The file name of the input file.

intype The file type of the input file.

inmode
The file mode of the input file.

outname
The file name of the output file. If = is specified, the output file is the same as the input file.

outtype
The file type of the output file. If = is specified, the output file type is the same as the input file type.

outmode
The file mode of the output file. If = is specified, the output file mode is the same as the input filemode.

fromcode
The name of the codeset in which the input data is encoded.

tocode
The name of the codeset to which the output data is to be converted.

In the following example, the input file is `INPUT FILE A` in code page IBM-037 and the output file is `OUTPUT FILE A` in code page IBM-1047.

```
ICONV INPUT FILE A OUTPUT FILE A (FROMCODE IBM-037 TOCODE IBM-1047
```

Note: If the `FROMCODE` or `TOCODE` is specified more than once, the last value specified is used. The output file is created with a record format of V. For more information, refer to the *z/OS C/C++ Programming Guide*

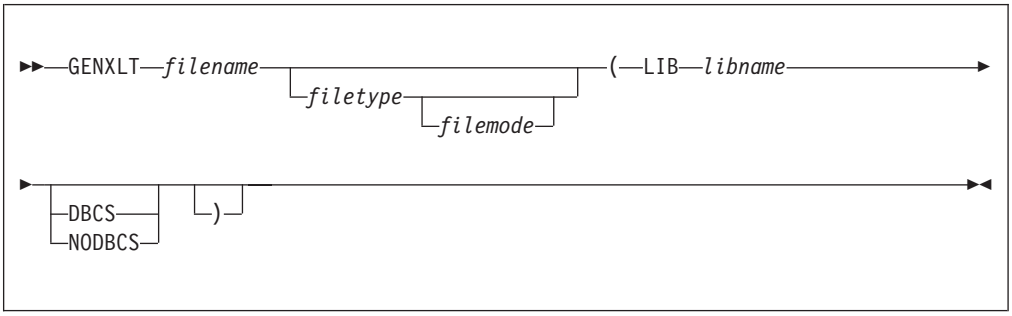
genxlt Utility

The `genxlt` utility creates translation tables, which are used by the `iconv_open()`, `iconv()`, and `iconv_close()` services of the run-time library. These services can be called from both non-XPLINK and XPLINK applications. The non-XPLINK and XPLINK versions have different names. The non-XPLINK version of the GENXLT table should always be generated. If any XPLINK applications will require one of these translation tables, then the XPLINK version should also be generated.

The `genxlt` utility reads character conversion information from the input file and writes the compiled conversion table to the LOADLIB. The input file contains directives that are acted upon by the `genxlt` utility to produce the compiled version of the conversion table. The source input to the `genxlt` utility is assumed to be implicitly specified in code page IBM-1047.

A REXX EXEC, `GENXLT`, invokes the `genxlt` utility to read the character conversion information and produces the conversion table. It may be invoked under VM/CMS or VM batch. The `genxlt` utility options can be specified on the command line. If the same option is specified more than once, the last option specified is used.

The general parameters of `GENXLT` EXEC are as follows:



Where:

filename
The file name for the file containing the character conversion information.

filetype
The file type for the file containing the character conversion information. If it is not specified, it defaults to `GENXLT`.

filemode
The file mode for the file containing the character conversion information. If it is not specified, the accessed disks are searched for the first file that matches the file name and file type.

libname
The `libname` parameter specifies the name of the LOADLIB. The member name in the LOADLIB will be the same as `filename`.

DBCS|NODBCS
Specifies whether the DBCS characters within shift-out and shift-in characters will be converted. The `DBCS` option should only be specified when an EBCDIC code page is being converted to a different EBCDIC code page.

If the `DBCS` option is specified, when a shift-out character is encountered in the input, the characters up to the shift-in character are copied to the output, and not converted. There must be an even number of characters between the shift-out and shift-in characters, and the characters must be valid DBCS characters.

If the `NODBCS` option is specified (or by default), all the characters are converted, and no checking of DBCS characters is performed.

For more information, refer to the *z/OS C/C++ Programming Guide*.

The conversion table is built as a member of the loadlib specified. The member name is the same as the `filename` specified.

In the following example, the input file is `EDCUEAEY GENXLT A`, the library is `MYLIB` with the `DBCS` option, and the conversion is from `IBM-037` to `IBM-1047`.

```
GENXLT EDCUEAEY GENXLT A (LIB MYLIB DBCS
```

To make the conversion table available for the `iconv` utility and `iconv_open()` function, issue the GLOBAL LOADLIB command, as follows:

```
GLOBAL LOADLIB MYLIB SCEERUN
```

localedef Utility

The `localedef` utility creates locale objects, which are used by the `setlocale()` service of the run-time library. This service can be called from both non-XPLINK and XPLINK applications. The non-XPLINK and XPLINK locale object versions have different names. The non-XPLINK version of the locale object should always be generated. If any XPLINK applications will use the locale then the XPLINK version should also be generated.

A *locale* is a collection of data that defines language and cultural conventions. Locales consist of various categories, that are identified by name, that characterize specific aspects of your cultural environment.

The `localedef` utility generates locales according to the rules that are defined in the locale definition file. A user can create his own customized locale definition file.

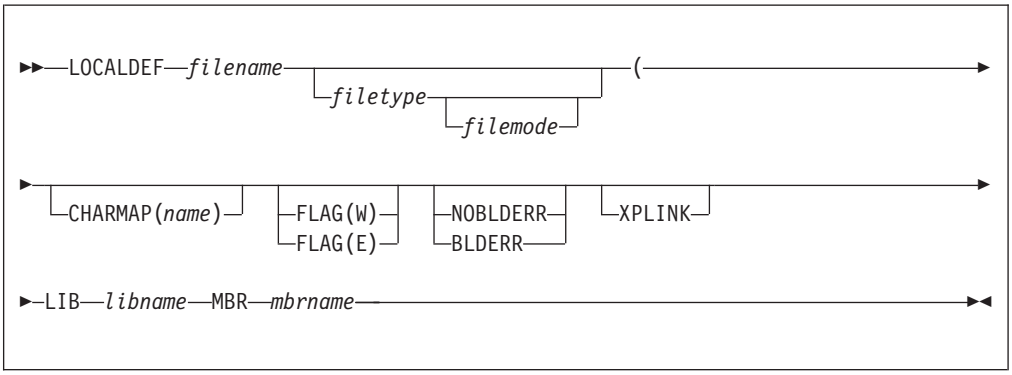
The `localedef` utility reads the locale definition file and produces a locale object that can be used by the locale specific library functions.

A REXX EXEC `LOCALDEF` invokes the `localedef` utility under VM/CMS and VM batch. It does the following:

1. Invokes the CCNELDEF module to read the locale definition file and produce the C code to build the locale
2. Invokes the C/C++ for z/VM compiler to compile the C source generated
3. Invokes the VM/CMS BIND command to build a loadlib member

The options for the localedef utility are specified on the command line. They can be separated by spaces or commas. If the same option is specified more than once, the last option specified is used.

The general form of the LOCALDEF EXEC is as follows:



Where:

filename
The file name for the file containing the locale definition information.

filetype
The file type for the file containing the locale definition information. If it is not specified, it defaults to LOCALE.

filemode
The file mode for the file containing the locale definition information. If it is not specified, the accessed disks are searched for the first file that matches the file name and file type.

CHARMAP(name)
Specifies the member name of the file containing the mapping of the character symbols to actual character encodings. If this option is not specified, the localedef utility defaults the Charmap to IBM-1047.

The name specified is the file name of the charmap file. The file type is CHARMAP.

FLAG(W|E)
The FLAG option controls whether warning messages are issued. If FLAG(W) is specified (or by default), warning and error messages are issued. If FLAG(E) is specified, only the error messages are issued.

BLDERR|NOBLDERR
If the BLDERR option is specified, the locale is generated even if errors are detected. If the NOBLDERR option is specified (or by default), the locale is not generated if an error is detected.

XPLINK
Indicates that the locale to be built is an XPLINK locale.

libname

The `libname` parameter of the LIB option that specifies the name of the LOADLIB.

mbrname

The `mbrname` parameter of the MBR option specifies the member name for the member in the LOADLIB. The member name defaults to the file name of the input file.

The LOADLIB member is created using the BIND command. The member name within the LOADLIB is the member name (if specified) or the file name of the input file. The non-XPLINK version of the locale object should have EDC$ or EDC@ as the first four characters of the member name. The XPLINK version should have CEH$ or CEH@ as the first four characters of the member name.

For more information on locale and code set codes, refer to the *z/OS C/C++ Programming Guide*.

In the following example, the locale source is `EDC$EUEY LOCALE A`, the library name is `MYLIB`, options are `CHARMAP(IBM-297)`, and the output member name is `EDC$EUEM`, for `EN_US.IBM-297`.

```
LOCALDEF EDC$EUEY LOCALE A (LIB MYLIB CHARMAP(IBM-297) MBR EDC$EUEM
```

Chapter 6. z/VM OpenExtensions: Utilities

ar and make Utlities

This chapter describes the z/VM OpenExtensions `ar` and `make` utilities.

The z/VM OpenExtensions Shell and Utilities feature provides two utilities that you can use to make the task of creating and managing OpenExtensions C/C++ for z/VM application programs easier: `ar` and `make`. Use these utilities with the `c89/cxx` utility to build an application program into an easily updated and maintained executable file.

Note: All references to `c89` in the following sections also apply to `cxx` unless otherwise specified.

z/VM OpenExtensions Archive Libraries

The `ar` utility allows you to create and maintain a library of z/VM OpenExtensions C/C++ for z/VM application object files. You can specify the `c89` command string so that archive libraries are processed during binding.

The archive library file, when created for application program object files, has a special symbol table for members that are object files. The symbol table is read to determine which object files should be bound into the application program executable file. A `c89`-specified archive library is processed during binding. Any object files in the specified archive library will be bound if they can be used to resolve external symbols. Use of this autocall library mechanism by the `c89` utility is analogous to the use of the `C370LIB` Object Library utility for z/VM application program objects. For more information, see “Object Library Utility” on page 87.

The `c89` utility requires that archive libraries obey the following naming convention in the byte file system (BFS):

filename`.a` This assumes that no directory file searching for the archive file takes place when specified on the `c89` command line. For example, to compile the application program source file `dirsum.c` from the `src` subdirectory of your working directory and resolve externals symbols from the `symb.a` archive library in your working directory, you would enter:

```
c89 -o ./exfils/dirsum ./src/dirsum.c ./symb.a
```

To use `c89` to search for specified archive files in one or more BFS directories, use the naming convention:

`lib`*libname*`.a` On the `c89` command line, specify BFS directories to be searched with the -L *directory* option and an archive library with the `-l` *libname* operand. For example, to compile the application program source file `entinfo.c` from the `src` subdirectory of your working directory and bind it with the object file `newsroute.o` and the archive file `/mylib/libbrwobjs.a`, enter:

```
c89 -o ./entinfo -L /mylib ./src/entinfo.c newsroute.o -l brwobjs
```

The BFS subdirectory `mylib` of your working directory is searched first for the archive library `libbrwobjs.a`. If it is not found there, `c89` searches for the archive library in the usual places.

Creating Archive Libraries

To create the archive library, use the `ar -r` option. For example, to create an archive library named `bin/libbrobompgm.a` from your working directory and add the member `jkeyadd.o` to it, specify:

```
ar -rc ./bin/libbrobompgm.a jkeyadd.o
```

The `libbrobompgm.a` archive library file is created in the `bin` subdirectory of your BFS working directory. Use of the `-c` option tells `ar` to suppress the message normally sent when an archive library file is created.

To display the object files archived in the `bin/libbrobompgm.a` library from your working directory, specify:

```
ar -t ./bin/libbrobompgm.a
```

For a detailed discussion of the z/VM OpenExtensions VM `ar` utility, see *z/VM OpenExtensions Command Reference*.

Creating Makefiles

The `make` utility maintains all the parts of and dependencies for your application program. It uses a *makefile*, which you create, to keep your application parts (listed in it) up to date with one another. If one part changes, `make` updates all the other files that depend on the changed part.

A makefile is a normal BFS text file. Create the file and edit it using any text editor to describe the application program files, their locations, dependencies on other files, and rules for building the files into an application executable file. When creating a makefile, remember that tabbing of information in the file is important and not all editors support tab characters the same way.

The `make` utility invokes the `c89` interface to the C/C++ for z/VM compiler and the binder to recompile and bind an updated application program.

For a detailed discussion of the shell `make` utility and how best take advantage of its function, see *z/VM OpenExtensions Advanced Application Programming Tools* and *z/VM OpenExtensions Command Reference*.

Appendix A. IBM Supplied EXECs

This appendix lists the EXECs provided by the C/C++ for z/VM compiler in conjunction with the Language Environment, to call the various utilities. For more information on the EXECs provided by Language Environment see the *Language Environment Programming Guide*.

EXEC Name	Task Description
CC	Compile
CMOD	Generate an executable module
CXXFILT	Demangle names
GENXLT	Generate a translate table for use by ICONV utility and functions
ICONV	Convert a file from one code set encoding to another
LOCALDEF	Produce a locale object that can be used by the locale specific library functions
LINKLOAD	Generate a fetchable module
C370LIB	Maintain an object library TXTLIB
CDSECT	Run the DSECT Conversion Utility

Appendix B. C/C++ for z/VM Compiler Return Codes and Messages

See the *z/OS C/C++ Messages* for complete descriptions of C/C++ for z/VM return codes and messages.

Appendix C. EXEC Error Messages

The messages in this section can be returned from the following C/C++ for z/VM EXECs:

- CC
- CDSECT
- LOCALDEF

Message Format:

CCNUTLnnnx text <&n> where:

nnn - error message number
x - can be one of I for Informational, W for Warning, or E for Error
text - message which appears on the screen
&***n*** - substitution variable

CCNUTL001I &1 exec completed with return code &2.

Explanation: The utility completed processing with the return code specified.

User Response: No response required.

CCNUTL002E Help is not available.

Explanation: The help file for the requested command is not accessible or does not exist.

User Response: Find out from your systems programmer which disk has the help file on it and get access to the disk. If the help file has not been installed, have your systems programmer install it.

CCNUTL003W LE Run-Time library SCEERUN is not in GLOBAL LOADLIB.

Explanation: The Run-Time library is missing.

User Response: Run the VM command GLOBAL LOADLIB SCEERUN to add the Run-Time library.

CCNUTL004W Invalid parameter list.

Explanation: The parameter list is not valid.

User Response: Check the syntax of the command you are running and correct it.

CCNUTL005E A-Disk is not accessed.

Explanation: Your A-disk is not accessed in read/write mode.

User Response: Link to and access your A-disk in read/write mode.

CCNUTL006E A-Disk is not writable.

Explanation: Your A-disk is not accessed in read/write mode.

User Response: Link to and access your A-disk in read/write mode.

CCNUTL008E A library name must be specified in suboption LIB.

Explanation: You must specify a library name when using LOCALDEF command.

User Response: Check the syntax of the LOCALDEF command and correct it.

CCNUTL009E Cannot execute program module &1.

Explanation: A module cannot be run.

User Response: Check with your system programmer.

Appendix D. Run-Time Error Messages and Return Codes

This appendix contains information about the Run-Time messages and should not be used as programming interface information.

These are messages you see while your C/C++ program is running. Messages may be displayed in uppercase or in mixed case English format, or in Kanji.

perror Messages

These messages are only printed when a call to `perror` or `strerror` is made and the `errno` value does not prefix the message.

Note: Information about these messages is found in *Language Environment Run-Time Messages*.

C/C++ for z/VM Run-Time Return Codes

The Run-Time return code value is set in one of the following ways:

- By the initialization and termination routines or the program management routines of Language Environment.
- By the `return` statement in your C/C++ program
- By calling the `exit` or `abort` functions from your C/C++ program.

It is possible to pass a return code from a C/C++ program to the program that invoked it. For example, if the C/C++ program is invoked by a REXX EXEC, it can examine the return code to determine if processing should continue.

The return code generated by a C/C++ program consists of two elements. One element is specified if the program calls the `exit` function or if the program specifies a return value when returning from `main`. The other element is specified by the program management routines of the Language Environment library and indicates the way in which your program terminated. Unless an error is detected that prevents the program management routines from operating correctly, the two elements are added together to form a total in which the thousands digit indicates the way in which your program terminated and the hundreds, tens, and units are set by your program.

Valid return codes are -2^{31} to $2^{31}-1$, inclusive.

Note: The CMS READY(xxxxx) prompt displays only the last 5 digits of the return code. For example, 2,000,000 is displayed as `READY(00000)`. You can write a REXX EXEC to retrieve the full return code.

See the *Language Environment Run-Time Messages* for a list of error messages.

Appendix E. Utility Messages

This appendix contains information about the DSECT utility messages. See the *Language Environment Run-Time Messages* for messages and return codes for the following:

- Object Library Utility
- Run-time messages and return codes
- `localedef` Utility
- `genxlt` Utility
- `iconv` Utility

See the *z/OS C/C++ Messages* for messages and return codes for the CXXFILT utility.

DSECT Utility Messages

Return Codes

Table 8. Return Codes from the DSECT Utility

Return Code	Meaning
0	Successful completion.
4	Successful completion, warnings issued.
8	DSECT Utility failed, error messages issued.
12	DSECT Utility failed, severe error messages issued.
16	DSECT Utility failed, insufficient storage to continue processing.

Messages

The messages issued by the `DSECT` utility have the format **EDCnnnn ss text <&x>** where:

nnnn Error message number

ss Error severity, will have one of the following values:

- **00** Informational message
- **10 or E** Error warning message
- **30** Error message
- **40** Severe error message

text Message which appears on the screen

&x Substitution variable

The messages that may be issued are as follows:

EDC5500 10 Option %s is not valid and is ignored.

Explanation: The option specified in the message is not valid DSECT Utility option or a valid option has been specified with an invalid value. The specified option is ignored.

User Response: Rerun the DSECT Utility with the correct option.

EDC5501 30 No DSECT or CSECT names were found in the SYSADATA file.

Explanation: The SECT option was not specified or SECT(ALL) was specified. The SYSADATA was searched for all DSECTs and CSECTs but no DSECTs or CSECTs were found.

User Response: Rerun the DSECT Utility with a SYSADATA file that contains the required DSECT or CSECT definition.

Severity: 30

EDC5502 30 Sub option %s for option %s is too long.

Explanation: The sub option specified for the option was too long and is ignored.

EDC5503 30 Section name %s was not found in SYSADATA File.

Explanation: The section name specified with the SECT option was not found in the External Symbol records in the SYSADATA file. The C structure is not produced.

User Response: Rerun the DSECT Utility with a SYSADATA file that contains the required DSECT or CSECT definition.

EDC5504 30 Section name %s is not a DSECT or CSECT.

Explanation: The section name specified with the SECT option is not a DSECT or CSECT. Only a DSECT or CSECT names may be specified. The C structure is not produced.

EDC5505 00 No fields were found for section %s, structure is not produced.

Explanation: No field records were found in the SYSADATA file that matched the ESDID of the specified section name. The C structure is not produced.

EDC5506 30 Record length for file ″%s″ is too small for the SEQUENCE option, option ignored.

Explanation: The record length for the output file specified is too small to enable the SEQUENCE option to generate the sequence number in columns 73 to 80. The available record length must be greater than or equal to 80 characters. The SEQUENCE option is ignored.

EDC5507 40 Insufficient storage to continue processing.

Explanation: No further storage was available to continue processing.

User Response: Rerun the DSECT Utility with a larger virtual machine (CMS).

EDC5508 30 Open failed for file ″%s″: %s

Explanation: This message is issued if the open fails for any file required by the DSECT Utility. The file name passed to fopen() and the error message returned by strerror(errno) is included in the message.

User Response: The message text indicates the cause of the error. If the file name was specified incorrectly on the OUTPUT option, rerun the DSECT Utility with the correct file name.

EDC5509 40 %s failed for file ″%s″: %s

Explanation: This message is issued if any error occurs reading, writing or positioning on any file by the DSECT Utility. The name of the function that failed (Read, Write, fgetpos, fsetpos), file name and text from strerror(errno) is included in the message.

User Response: This message may be issued if an error occurs reading or writing to a file. This may be caused by an error within the file, such as an I/O error or insufficient disk space. Correct the error and rerun the DSECT Utility.

EDC5510 40 Internal Logic error in function %s

Explanation: The DSECT Utility has detected that an error has occurred while generating the C structure. Processing is terminated and the C structure is not produced.

User Response: This may be caused by an error in the DSECT Utility or by incorrect input in the SYSADATA file. Contact your systems administrator.

EDC5511 10 No matching right parenthesis for %s option.

Explanation: The option specified had a sub option beginning with a left parenthesis but no right parenthesis was present.

User Response: Rerun the DSECT Utility with the parenthesis for the option correctly paired.

EDC5512 10 No matching quote for %s option.

Explanation: The OUTPUT option has a sub option beginning with a single quote but no matching quote was found.

User Response: Rerun the DSECT Utility with the

quotes for the option correctly paired.

EDC5513 10 Record length too small for file ″%s″.

Explanation: The record length for the Output file specified is less than 10 characters in length. The minimum available record length must be at least 10 characters.

User Response: Rerun the DSECT Utility with an output file with a available record length of at least 10 characters.

EDC5514 30 Too many sub options were specified for option %s.

Explanation: More than the maximum number of sub options were specified for the particular option. The extra sub options are ignored.

EDC5515 00 HDRSKIP option value greater than length for section %s, structure is not produced.

Explanation: The value specified for the HDRSKIP option was greater than the length of the section. A structure was not produced for the specified section.

User Response: Rerun the DSECT Utility with a smaller value for the HDRSKIP option.

EDC5516 10 SECT and OPTFILE options are mutually exclusive, OPTFILE option is ignored

Explanation: Both the SECT and OPTFILE options were specified, but the options are mutually exclusive.

User Response: Rerun the DSECT Utility with either the SECT or OPTFILE option.

EDC5517 10 Line %i from ″%s″ does not begin with SECT option

Explanation: The line from the file specified on the OPTFILE option did not begin with the SECT option. The line was ignored.

User Response: Rerun the DSECT Utility without OPTFILE option, or correct the line in the input file.

EDC5518 10 setlocale() failed for locale name ″%s″.

Explanation: The setlocale() function failed with the locale name specified on the LOCALE option. The LOCALE option was ignored.

User Response: Rerun the DSECT Utility without LOCALE option, or correct the locale name specified with the LOCALE option.

Appendix F. Layout of the Events File

This appendix specifies the layout of the SYSEVENT file. The SYSEVENT file contains error information and source file statistics. Use the EVENTS compiler option to produce the SYSEVENT file. For more information on the EVENTS compiler option, see “EVENTS | NOEVENTS” on page 41.

In the following example, the source file SIMPLE C is compiled with the EVENTS(EGEVENT FILE) compiler option. The file ERR H is a header file that is included in SIMPLE C. Figure 39 is the event file that is generated when SIMPLE C is compiled.

```
1   #include "err.h"
2   main() {
3      add some error messages;
4      return(0);
5      here and there;
6   }
```

Figure 37. SIMPLE C

```
1   add some;
2   errors in the header file;
```

Figure 38. ERR H

```
-------  start simple.events  ------
 FILEID 0 1 0 13 'SIMPLE C A1'
 FILEID 0 2 1 8 ERR H A1
 ERROR 0 2 1 0 1 1 1 8 CCN3166 E 12 48 Definition of function add requires parentheses.
 FILEEND 0 2 2
 ERROR 0 2 1 0 1 5 2 8 CCN3276 E 12 35 Syntax error: possible missing '{'?
 ERROR 0 1 1 0 3 4 3 27 CCN3045 E 12 26 Undeclared identifier add.
 ERROR 0 1 1 0 5 9 5 18 CCN3277 E 12 42 Syntax error: possible missing ';' or ','?
 ERROR 0 1 1 0 5 4 5 18 CCN3045 E 12 27 Undeclared identifier here.
 FILEEND 0 1 6
-------  end   simple.events  ------
```

Figure 39. Sample SYSEVENT file

There are three different record types generated in the event file:

- FILEID
- FILEEND
- ERROR

Description of the Fileid Field

The following is an example of the FILEID field from the sample SYSEVENT file that is shown in Figure 39. Table 9 on page 130 describes the FILEID identifiers.

```
 FILEID 0 1 0 13 'SIMPLE C A1'
        A B C D  E
```

Table 9. Explanation of the FILEID Field Layout

Column	Identifier	Description
A	Revision	Revision number of the event record.
B	File number	Increments starting with 1 for the primary file.
C	Line number	The line number of the `#include` directive. For the primary source file, this value is 0.
D	File name length	Length of file or data set.
E	File name	String containing file/data set name.

Description of the Fileend Field

The following is an example of the FILEEND field from the sample SYSEVENT file that is shown in Figure 39 on page 129. Table 10 describes the FILEEND identifiers.

```
FILEEND 0 1 6
        A B C
```

Table 10. Explanation of the FILEEND Field Layout

Column	Identifier	Description
A	Revision	Revision number of the event record.
B	File number	File number that has been processed to end of file.
C	Expansion	Total number of lines in the file.

Description of the Error Field

The following is an example of the ERROR field from the sample SYSEVENT file that is shown in Figure 39 on page 129. Table 11 describes the ERROR identifiers.

```
ERROR 0 1 1 0 3 4 3 27 CCN3045 E 12 26 Undeclared identifier add.
      A B C D E F G H  I       J K  L  M
```

Table 11. Explanation of the ERROR Field Layout

Column	Identifier	Description
A	Revision	Revision number of the event record.
B	File number	Increments starting with 1 for the primary file.
C	Reserved	Do not build a dependency on this identifier. It is reserved for future use.
D	Reserved	Do not build a dependency on this identifier. It is reserved for future use.
E	Starting line number	The source line number for which the message was issued. A value of 0 indicates the message was not associated with a line number.
F	Starting column number	The column number or position within the source line for which the message was issued. A value of 0 indicates the message is not associated with a line number.
G	Reserved	Do not build a dependency on this identifier. It is reserved for future use.
H	Reserved	Do not build a dependency on this identifier. It is reserved for future use.

Table 11. Explanation of the ERROR Field Layout (continued)

Column	Identifier	Description
I	Message identifier	String Containing the message identifier.
J	Message severity character	I=Informational W=Warning E=Error S=Severe U=Unrecoverable
K	Message severity number	Return code associated with the message.
L	Message length	Length of message text.
M	Message text	String containing message text.

Notices

Any reference to an IBM licensed program in this publication is not intended to state or imply that only IBM's licensed program may be used. Any functionally equivalent product, program or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

IBM may have patents or pending patent applications covering subject matter 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 the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY, 10594, USA.

This publication contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples 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.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independent created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact the IBM Corporation, Mail Station P300, 522 South Road, Poughkeepsie, NY 12601-5400, USA, Attention: Information Request. Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

Any pointers in this publication to Web sites are provided for convenience only and do not in any manner serve as an endorsement of these Web sites.

Programming Interface Information

This publication documents *intended* Programming interfaces that allow the customer to write C/C++ for z/VM programs.

Trademarks

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

ESA/390	IBM	Language Environment
MVS	OS/390	SOM
S/390	System/390	VM/ESA

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

Glossary

This glossary defines technical terms and abbreviations that are used in z/OS C/C++ and C/C++ for z/VM documentation. If you do not find the term you are looking for, refer to the index of the appropriate manual or view *IBM Glossary of Computing Terms*, located at:

`http://www.ibm.com/ibm/terminology/goc/gocmain.htm`

This glossary includes terms and definitions from:

- *American National Standard Dictionary for Information Systems*, ANSI/ISO X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI/ISO). Copies may be purchased from the American National Standards Institute, 11 West 42nd Street, New York, New York 10036. Definitions are indicated by the symbol *ANSI/ISO* after the definition.
- *IBM Dictionary of Computing*, SC20-1699. These definitions are indicated by the registered trademark *IBM.* after the definition.
- *X/Open CAE Specification, Commands and Utilities, Issue 4. July, 1992*. These definitions are indicated by the symbol *X/Open* after the definition.
- *ISO/IEC 9945-1:1990/IEEE POSIX 1003.1-1990*. These definitions are indicated by the symbol *ISO.1* after the definition.
- The *Information Technology Vocabulary*, developed by Subcommittee 1, Joint Technical Committee 1, of the International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC JTC1/SC1). Definitions of published parts of this vocabulary are identified by the symbol *ISO-JTC1* after the definition; definitions taken from draft international standards, committee drafts, and working papers being developed by ISO/IEC JTC1/SC1 are identified by the symbol *ISO Draft* after the definition, indicating that final agreement has not yet been reached among the participating National Bodies of SC1.

A

abstract class. (1) A class with at least one pure virtual function that is used as a *base class* for other classes. The abstract class represents a concept; classes derived from it represent implementations of the concept. You cannot create a direct object of an abstract class, but you can create references and pointers to an abstract class and set them to refer to objects of classes derived from the abstract class. See also *base class*. (2) A class that allows polymorphism. There can be no objects of an abstract class; they are only used to derive new classes.

abstract code unit. See *ACU*

abstract data type. A mathematical model that includes a structure for storing data and operations that can be performed on that data. Common abstract data types include sets, trees, and heaps.

abstraction (data). A data type with a private representation and a public set of operations (functions or operators) which restrict access to that data type to that set of operations. The C++ language uses the concept of classes to implement data abstraction.

access. An attribute that determines whether or not a class member is accessible in an expression or declaration.

access declaration. A declaration used to restore access to members of a base class.

access mode. (1) A technique that is used to obtain a particular logical record from, or to place a particular logical record into, a file assigned to a mass storage device. *ANSI/ISO.* (2) The manner in which files are referred to by a computer. Access can be sequential (records are referred to one after another in the order in which they appear on the file), access can be random (the individual records can be referred to in a nonsequential manner), or access can be dynamic (records can be accessed sequentially or randomly, depending on the form of the input/output request). *IBM.* (3) A particular form of access permitted to a file. *X/Open.*

access resolution. The process by which the accessibility of a particular class member is determined.

access specifier. One of the C++ keywords: public, private, and protected, used to define the access to a member.

ACU (abstract code unit). A measurement used by the C/C++ compiler for judging the size of a function. The number of ACUs that comprise a function is proportional to its size and complexity.

additional heap. A Language Environment heap created and controlled by a call to CEECRHP. See also *below heap*, *anywhere heap*, and *initial heap*.

addressing mode. See *AMODE*.

address space. (1) The range of addresses available to a computer program. *ANSI/ISO.* (2) The complete range of addresses that are available to a programmer. See also *virtual address space*. (3) The area of virtual

storage available for a particular job. The memory locations that can be referenced by a process. *X/Open. ISO.1.*

aggregate. (1) An array or a structure. (2) A compile-time option to show the layout of a structure or union in the listing. (3) In programming languages, a structured collection of data items that form a data type. *ISO-JTC1.* (4) In C++, an array or a class with no user-declared constructors, no private or protected non-static data members, no base classes, and no virtual functions.

alert. (1) A message sent to a management services focal point in a network to identify a problem or an impending problem. *IBM.* (2) To cause the user's terminal to give some audible or visual indication that an error or some other event has occurred. When the standard output is directed to a terminal device, the method for alerting the terminal user is unspecified. When the standard output is not directed to a terminal device, the alert is accomplished by writing the alert character to standard output (unless the utility description indicates that the use of standard output produces undefined results in this case). *X/Open.*

alert character. A character that in the output stream should cause a terminal to alert its user via a visual or audible notification. The alert character is the character designated by a '\a' in the C and C++ languages. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the alert function. *X/Open.*

This character is named <alert> in the portable character set.

alias. (1) An alternate label; for example, a label and one or more aliases may be used to refer to the same data element or point in a computer program. *ANSI/ISO.* (2) An alternate name for a member of a partitioned data set. *IBM.* (3) An alternate name used for a network. Synonymous with *nickname*. *IBM.*

alias name. (1) A word consisting solely of underscores, digits, and alphabetics from the portable file name character set, and any of the following characters: ! % , @. Implementations may allow other characters within alias names as an extension. *X/Open.* (2) An alternate name. *IBM.* (3) A name that is defined in one network to represent a logical unit name in another interconnected network. The alias name does not have to be the same as the real name. If these names are not the same, translation is required. *IBM.*

alignment. The storing of data in relation to certain machine-dependent boundaries. *IBM.*

alternate code point. A syntactic code point that permits a substitute code point to be used. For example, the left brace ({) can be represented by X'B0' and also by X'C0'.

American National Standard Code for Information Interchange (ASCII). The standard code, using a coded character set consisting of 7-bit coded characters (8 bits including parity check), that is used for information interchange among data processing systems, data communication systems, and associated equipment. The ASCII set consists of control characters and graphic characters. *IBM.*

Note: IBM has defined an extension to ASCII code (characters 128-255).

American National Standards Institute (ANSI/ISO). An organization consisting of producers, consumers, and general interest groups, that establishes the procedures by which accredited organizations create and maintain voluntary industry standards in the United States. *ANSI/ISO.*

AMODE (addressing mode). A program attribute that refers to the address length that a program is prepared to handle upon entry. Addresses may be 24 or 31 bits in length. *IBM.*

angle brackets. The characters < (left angle bracket) and > (right angle bracket). When used in the phrase ″enclosed in angle brackets,″ the symbol < immediately precedes the object to be enclosed, and > immediately follows it. When describing these characters in the portable character set, the names <less-than-sign> and <greater-than-sign> are used. *X/Open.*

anonymous union. A union that is declared within a structure or class and does not have a name. It must not be followed by a declarator.

ANSI/ISO. See *American National Standards Institute*.

ANSI C. A definition of the programming language developed by the American National Standards Institute. The standard document is designated by ANSI/ISO 9899-1990ì1992Ù. It was previously known as ANSI X3.159-1989.

anywhere heap. The C/C++ controlled by the ANYHEAP run-time option. It contains library data, such as run-time control blocks and data structures not normally accessible from user code. The anywhere heap may reside above 16M. See also *below heap*, *additional heap*, and *initial heap*.

API (application program interface). A functional interface supplied by the operating system or by a separately orderable licensed program that allows an application program written in a high-level language to use specific data or functions of the operating system or the licensed program. *IBM.*

application. (1) The use to which an information processing system is put; for example, a payroll application, an airline reservation application, a network application. *IBM.* (2) A collection of software

components used to perform specific types of user-oriented work on a computer. *IBM.*

application generator. An application development tool that creates applications, application components (panels, data, databases, logic, interfaces to system services), or complete application systems from design specifications.

application program. A program written for or by a user that applies to the user's work, such as a program that does inventory control or payroll. *IBM.*

archive libraries. The archive library file, when created for application program object files, has a special symbol table for members that are object files.

argument. (1) A parameter passed between a calling program and a called program. *IBM.* (2) In a function call, an expression that represents a value that the calling function passes to the function specified in the call. (3) In the shell, a parameter passed to a utility as the equivalent of a single string in the *argv* array created by one of the *exec* functions. An argument is one of the options, option-arguments, or operands following the command name. *X/Open.*

argument declaration. See *parameter declaration*.

arithmetic object. (1) A bit field, or an integral, floating-point, or packed decimal (IBM extension) object. (2) A real object or objects having the type float, double, or long double.

array. In programming languages, an aggregate that consists of data objects with identical attributes, each of which may be uniquely referenced by subscripting. *ISO-JTC1.*

array element. A data item in an array. *IBM.*

ASCII. See *American National Standard Code for Information Interchange*.

Assembler H. An IBM licensed program. Translates symbolic assembler language into binary machine language.

assembler language. A source language that includes symbolic language statements in which there is a one-to-one correspondence with the instruction formats and data formats of the computer. *IBM.*

assembler user exit. In Language Environment a routine to tailor the characteristics of an enclave prior to its establishment.

assignment expression. An expression that assigns the value of the right operand expression to the left operand variable and has as its value the value of the right operand. *IBM.*

atexit list. A list of actions specified in the C/C++ `atexit()` function that occur at normal program termination.

auto storage class specifier. A specifier that enables the programmer to define a variable with automatic storage; its scope restricted to the current block.

automatic call library. Contains modules that are used as secondary input to the binder to resolve external symbols left undefined after all the primary input has been processed.

The automatic call library can contain:

- Object modules, with or without binder control statements
- Load modules
- C/C++ run-time routines (SCEELKED)

automatic library call. The process in which control sections are processed by the binder or loader to resolve references to members of partitioned data sets. *IBM.*

automatic storage. Storage that is allocated on entry to a routine or block and is freed on the subsequent return. Sometimes referred to as *stack storage* or *dynamic storage*.

B

background job. (1) A low-priority job, usually a batched or noninteractive job. *IBM.* (2) A background process group. *X/Open.*

background process. (1) A process that does not require operator intervention but can be run by the computer while the workstation is used to do other work. *IBM.* (2) A mode of program execution in which the shell does not wait for program completion before prompting the user for another command. *IBM.* (3) A process that is a member of a background process group. *X/Open. ISO.1.*

background process group. Any process group, other than a foreground process group, that is a member of a session that has established a connection with a controlling terminal. *X/Open. ISO.1.*

backslash. The character \. This character is named <backslash> in the portable character set.

base class. A class from which other classes are derived. A base class may itself be derived from another base class. See also *abstract class*.

based on. The use of existing classes for implementing new classes.

below heap. The C/C++ heap controlled by the BELOWHEAP Run-Time option, which contains library data, such as run-time control block and data structures

not normally accessible from user code. Below heap always resides below 16M. See also *anywhere heap*, *initial heap*, and *additional heap*.

binary expression. An expression containing two operands and one operator.

binary stream. (1) An ordered sequence of untranslated characters. (2) A sequence of characters that corresponds on a one-to-one basis with the characters in the file. No character translation is performed on binary streams. *IBM.*

bind. (1) To combine one or more control sections or program modules into a single program module, resolving references between them. (2) To assign virtual storage addresses to external symbols.

binder. The DFSMS/MVS program that processes the output of language translators and compilers into an executable program (load module or program object). It replaces the linkage editor and batch loader.

bit field. A member of a structure or union that contains a specified number of bits. *IBM.*

bitwise operator. An operator that manipulates the value of an object at the bit level.

blank character. (1) A graphic representation of the space character. *ANSI/ISO.* (2) A character that represents an empty position in a graphic character string. *ISO Draft.* (3) One of the characters that belong to the *blank* character class as defined via the LC_CTYPE category in the current locale. In the POSIX locale, a blank character is either a tab or a space character. *X/Open.*

block. (1) In programming languages, a compound statement that coincides with the scope of at least one of the declarations contained within it. A block may also specify storage allocation or segment programs for other purposes. *ISO-JTC1.* (2) A string of data elements recorded or transmitted as a unit. The elements may be characters, words or physical records. *ISO Draft.* (3) The unit of data transmitted to and from a device. Each block contains one record, part of a record, or several records.

block statement. In the C or C++ languages, a group of data definitions, declarations, and statements appearing between a left brace and a right brace that are processed as a unit. The block statement is considered to be a single C or C++ statement. *IBM.*

boundary alignment. The position in main storage of a fixed-length field, such as a halfword or doubleword, on a byte-level boundary for that unit of information. *IBM.*

braces. The characters { (left brace) and } (right brace), also known as *curly* braces. When used in the phrase “enclosed in (curly) braces” the symbol { immediately precedes the object to be enclosed, and } immediately follows it. When describing these characters in the portable character set, the names <left-brace> and <right-brace> are used. *X/Open.*

brackets. The characters [(left bracket) and] (right bracket), also known as *square brackets*. When used in the phrase “enclosed in (square) brackets” the symbol [immediately precedes the object to be enclosed, and] immediately follows it. When describing these characters in the portable character set, the names <left-bracket> and <right-bracket> are used. *X/Open.*

break statement. A C or C++ control statement that contains the keyword *break* and a semicolon. *IBM.* It is used to end an iterative or a switch statement by exiting from it at any point other than the logical end. Control is passed to the first statement after the iteration or switch statement.

built-in. (1) A function that the compiler will automatically inline instead of making the function call, unless the programmer specifies not to inline. (2) In programming languages, pertaining to a language object that is declared by the definition of the programming language; for example, the built-in function SIN in PL/I, the predefined data type INTEGER in FORTRAN. *ISO-JTC1.* Synonymous with *predefined*. *IBM.*

byte-oriented stream. See *orientation of a stream*.

C

C library. A system library that contains common C language subroutines for file access, string operators, character operations, memory allocation, and other functions. *IBM.*

C or C++ language statement. A C or C++ language statement contains zero or more expressions. A block statement begins with a { (left brace) symbol, ends with a } (right brace) symbol, and contains any number of statements.

All C or C++ language statements, except block statements, end with a ; (semicolon) symbol.

C++ class library. A collection of C++ classes.

C++ library. A system library that contains common C++ language subroutines for file access, memory allocation, and other functions.

call. To transfer control to a procedure, program, routine, or subroutine. *IBM.*

call chain. A trace of all active functions.

callable services. A set of services that can be invoked by Language Environment-conforming high level languages using the conventional Language

Environment-defined call interface, and usable by all programs sharing the Language Environment conventions.

Use of these services helps to decrease an application's dependence on the specific form and content of the services delivered by any single operating system.

caller. A function that calls another function.

cancelability point. A specific point within the current thread that is enabled to solicit cancel requests. This is accomplished using the `pthread_testintr()` function.

carriage-return character. A character that in the output stream indicates that printing should start at the beginning of the same physical line in which the carriage-return character occurred. The carriage-return is the character designated by '\r' in the C and C++ languages. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the movement to the beginning of the line. *X/Open.*

CASE (Computer-Aided Software Engineering). A set of tools or programs to help develop complex applications. *IBM.*

case clause. In a C or C++ switch statement, a CASE label followed by any number of statements.

case label. The word `case` followed by a constant integral expression and a colon. When the selector evaluates the value of the constant expression, the statements following the case label are processed.

cast expression. An expression that converts or reinterprets its operand.

cast operator. The cast operator is used for explicit type conversions.

cataloged procedures. A set of control statements placed in a library and retrievable by name. *IBM.*

catch block. A block associated with a try block that receives control when an exception matching its argument is thrown.

char specifier. A char is a built-in data type. In the C++ language, char, signed char, and unsigned char are all distinct data types.

character. (1) A letter, digit, or other symbol that is used as part of the organization, control, or representation of data. A character is often in the form of a spatial arrangement of adjacent or connected strokes. *ANSI/ISO.* (2) A sequence of one or more bytes representing a single graphic symbol or control code. This term corresponds to the ISO C standard term *multibyte character* (multibyte character), where a single-byte character is a special case of the multibyte character. Unlike the usage in the ISO C standard, *character* here has no necessary relationship with storage space, and *byte* is used when storage space is discussed. *X/Open. ISO.1.*

character array. An array of type char. *X/Open.*

character class. A named set of characters sharing an attribute associated with the name of the class. The classes and the characters that they contain are dependent on the value of the LC_CTYPE category in the current locale. *X/Open.*

character constant. A string of any of the characters that can be represented, usually enclosed in quotes.

character set. (1) A finite set of different characters that is complete for a given purpose; for example, the character set in ISO Standard 646, 7-bit Coded Character Set for Information Processing Interchange. *ISO Draft.* (2) All the valid characters for a programming language or for a computer system. *IBM.* (3) A group of characters used for a specific reason; for example, the set of characters a printer can print. *IBM.* See also *portable character set*.

character special file. (1) A special file that provides access to an input or output device. The character interface is used for devices that do not use block I/O. *IBM.* (2) A file that refers to a device. One specific type of character special file is a terminal device file. *X/Open. ISO.1.*

character string. A contiguous sequence of characters terminated by and including the first null byte. *X/Open.*

child. A node that is subordinate to another node in a tree structure. Only the root node is not a child.

child enclave. The *nested enclave* created as a result of certain commands being issued from a *parent enclave*.

CICS (Customer Information Control System). Pertaining to an IBM licensed program that enables transactions entered at remote terminals to be processed concurrently by user-written application programs. It includes facilities for building, using, and maintaining databases. *IBM.*

class. (1) A C++ aggregate that may contain functions, types, and user-defined operators in addition to data. A class may be derived from another class, inheriting the properties of its parent class. A class may restrict access to its members. (2) A user-defined data type. A class data type can contain both data representations (data members) and functions (member functions).

class key. One of the C++ keywords: class, struct and union.

class library. A collection of classes.

class member operator. An operator used to access class members through class objects or pointers to class objects. The class member operators are:

```
.  ->  .*  ->*
```

class name. A unique identifier that names a class type.

class scope. An indication that a name of a class can be used only in a member function of that class.

class tag. Synonym for *class name*.

class template. A blueprint describing how a set of related classes can be constructed.

class template declaration. A class template declaration introduces the name of a class template and specifies its template parameter list. A class template declaration may optionally include a class template definition.

class template definition. A class template definition describes various characteristics of the class types that are its specializations. These characteristics include the names and types of data members of specializations, the signatures and definitions of member functions, accessibility of members, and base classes.

client program. A program that uses a class. The program is said to be a *client* of the class.

CMS. Conversational Monitor System.

CMS extended parameter list. A type of parameter list available in the CMS environment consisting of a string composed exactly as the user typed it at the terminal. There is no tokenization performed on the string.

CMS tokenized parameter list. A type of parameter list available in the CMS environment consisting of 8-byte tokens, folded to uppercase, terminating with a doubleword of X'FF'. Not supported under the C/C+.

COBOL (common business-oriented language). A high-level language, based on English, that is primarily used for business applications.

coded character set. (1) A set of graphic characters and their code point assignments. The set may contain fewer characters than the total number of possible characters: some code points may be unassigned. *IBM.* (2) A coded set whose elements are single characters; for example, all characters of an alphabet. *ISO Draft.* (3) Loosely, a code. *ANSI/ISO.*

code element set. (1) The result of applying a code to all elements of a coded set, for example, all the three-letter international representations of airport names. *ISO Draft.* (2) The result of applying rules that map a numeric code value to each element of a character set. An element of a character set may be related to more than one numeric code value but the reverse is not true. However, for state-dependent encodings the relationship between numeric code values to elements of a character set may be further controlled by state information. The character set may contain fewer elements than the total number of possible numeric code values; that is, some code values may be unassigned. *X/Open.* (3) Synonym for codeset.

code page. (1) An assignment of graphic characters and control function meanings to all code points; for example, assignment of characters and meanings to 256 code points for an 8-bit code, assignment of characters and meanings to 128 code points for a 7-bit code. (2) A particular assignment of hexadecimal identifiers to graphic characters.

code point. (1) A representation of a unique character. For example, in a single-byte character set each of 256 possible characters is represented by a one-byte code point. (2) An identifier in an alert description that represents a short unit of text. The code point is replaced with the text by an alert display program.

codeset. Synonym for code element set. *IBM.*

collating element. The smallest entity used to determine the logical ordering of character or wide-character strings. A collating element consists of either a single character, or two or more characters collating as a single entity. The value of the LC_COLLATE category in the current locale determines the current set of collating elements. *X/Open.*

collating sequence. (1) A specified arrangement used in sequencing. *ISO-JTC1. ANSI/ISO.* (2) An ordering assigned to a set of items, such that any two sets in that assigned order can be collated. *ANSI/ISO.* (3) The relative ordering of collating elements as determined by the setting of the LC_COLLATE category in the current locale. The character order, as defined for the LC_COLLATE category in the current locale, defines the relative order of all collating elements, such that each element occupies a unique position in the order. This is the order used in ranges of characters and collating elements in regular expressions and pattern matching. In addition, the definition of the collating weights of characters and collating elements uses collating elements to represent their respective positions within the collation sequence.

collation. The logical ordering of character or wide-character strings according to defined precedence rules. These rules identify a collation sequence between the collating elements, and such additional rules that can be used to order strings consisting or multiple collating elements. *X/Open.*

collection. (1) An abstract class without any ordering, element properties, or key properties. (2) In a general sense, an implementation of an abstract data type for storing elements.

Collection Class Library. A set of classes that provide basic functions for collections, and can be used as base classes.

column position. A unit of horizontal measure related to characters in a line.

It is assumed that each character in a character set has an intrinsic column width independent of any output device. Each printable character in the portable character set has a column width of one. The standard utilities, when used as described in this document set, assume that all characters have integral column widths. The column width of a character is not necessarily related to the internal representation of the character (numbers of bits or bytes).

The column position of a character in a line is defined as one plus the sum of the column widths of the preceding characters in the line. Column positions are numbered starting from 1. *X/Open.*

comma expression. An expression (not a function argument list) that contains two or more operands separated by commas. The compiler evaluates all operands in the order specified, discarding all but the last (rightmost). The value of the expression is the value of the rightmost operand. Typically this is done to produce side effects.

command. A request to perform an operation or run a program. When parameters, arguments, flags, or other operands are associated with a command, the resulting character string is a single command.

command processor parameter list (CPPL). The format of a TSO parameter list. When a TSO terminal monitor application attaches a command processor, register 1 contains a pointer to the CPPL, containing addresses required by the command processor.

Common Business-Oriented Language. See *COBOL*.

common expression elimination. Duplicated expressions are eliminated by using the result of the previous expression. This includes intermediate expressions within expressions.

compilation unit. (1) A portion of a computer program sufficiently complete to be compiled correctly. *IBM.* (2) A single compiled file and all its associated include files. (3) An independently compilable sequence of high-level language statements. Each high-level language product has different rules for what makes up a compilation unit.

complete class name. The complete qualification of a nested class name including all enclosing class names.

Complex Mathematics library. A C++ class library that provides the facilities to manipulate complex numbers and perform standard mathematical operations on them.

computational independence. No data modified by either a main task program or a parallel function is examined or modified by a parallel function that might be running simultaneously.

concrete class. (1) A class that is not abstract. (2) A class defining objects that can be created.

condition. (1) A relational expression that can be evaluated to a value of either true or false. *IBM.* (2) An exception that has been enabled, or recognized, by Language Environment and thus is eligible to activate user and language condition handlers. Any alteration to the normal programmed flow of an application. Conditions can be detected by the hardware/operating system and result in an interrupt. They can also be detected by language-specific generated code or language library code.

conditional expression. A compound expression that contains a condition (the first expression), an expression to be evaluated if the condition has a nonzero value (the second expression), and an expression to be evaluated if the condition has the value zero (the third expression).

condition handler. A user-written condition handler or language-specific condition handler (such as a PL/I ON-unit or C/C++ `signal()` function call) invoked by the *C/C++ condition manager* to respond to conditions.

condition manager. Manages conditions in the common execution environment by invoking various user-written and language-specific *condition handlers*.

condition token. In Language Environment, a data type consisting of 12 bytes (96 bits). The condition token contains structured fields that indicate various aspects of a condition including the severity, the associated message number, and information that is specific to a given instance of the condition.

const. (1) An attribute of a data object that declares the object cannot be changed. (2) A keyword that allows you to define a variable whose value does not change. (3) A keyword that allows you to define a parameter that is not changed by the function. (4) A keyword that allows you to define a member function that does not modify the state of the class for which it is defined.

constant. (1) In programming languages, a language object that takes only one specific value. *ISO-JTC1.* (2) A data item with a value that does not change. *IBM.*

constant expression. An expression having a value that can be determined during compilation and that does not change during the running of the program. *IBM.*

constant propagation. An optimization technique where constants used in an expression are combined and new ones are generated. Mode conversions are done to allow some intrinsic functions to be evaluated at compile time.

constructed reentrancy. The attribute of applications that contain external data and require additional processing to make them reentrant. Contrast with *natural reentrancy*.

constructor. A special C++ class member function that has the same name as the class and is used to create an object of that class.

control character. (1) A character whose occurrence in a particular context specifies a control function. *ISO Draft.* (2) Synonymous with non-printing character. *IBM.* (3) A character, other than a graphic character, that affects the recording, processing, transmission, or interpretation of text. *X/Open.*

control statement. (1) A statement that is used to alter the continuous sequential execution of statements; a control statement may be a conditional statement, such as if, or an imperative statement, such as return. (2) A statement that changes the path of execution.

controlling process. The session leader that establishes the connection to the controlling terminal. If the terminal ceases to be a controlling terminal for this session, the session leader ceases to be the controlling process. *X/Open. ISO.1.*

controlling terminal. A terminal that is associated with a session. Each session may have at most one controlling terminal associated with it, and a controlling terminal is associated with exactly one session. Certain input sequences from the controlling terminal cause signals to be sent to all processes in the process group associated with the controlling terminal. *X/Open. ISO.1.*

conversion. (1) In programming languages, the transformation between values that represent the same data item but belong to different data types. Information may be lost because of conversion since accuracy of data representation varies among different data types. *ISO-JTC1.* (2) The process of changing from one method of data processing to another or from one data processing system to another. *IBM.* (3) The process of changing from one form of representation to another; for example to change from decimal representation to binary representation. *IBM.* (4) A change in the type of a value. For example, when you add values having different data types, the compiler converts both values to a common form before adding the values.

conversion descriptor. A per-process unique value used to identify an open codeset conversion. *X/Open.*

conversion function. A member function that specifies a conversion from its class type to another type.

coordinated universal time (UTC). Synonym for Greenwich Mean Time (GMT). See *GMT*.

copy constructor. A constructor that copies a class object of the same class type.

CPPL. See *command processor parameter list*.

CSECT (control section). The part of a program specified by the programmer to be a relocatable unit, all elements of which are to be loaded into adjoining main storage locations.

Cross System Product. See *CSP*.

CSP (Cross System Product). A set of licensed programs designed to permit the user to develop and run applications using independently defined maps (display and printer formats), data items (records, working storage, files, and single items), and processes (logic). The Cross System Product set consists of two parts: Cross System Product/Application Development (CSP/AD) and Cross System Product/Application Execution (CSP/AE). *IBM.*

current working directory. (1) A directory, associated with a process, that is used in path name resolution for path names that do not begin with a slash. *X/Open. ISO.1.* (2) In the OS/2 operating system, the first directory in which the operating system looks for programs and files and stores temporary files and output. *IBM.* (3) In the z/VM OpenExtensions UNIX environment, a directory that is active and that can be displayed. Relative path name resolution begins in the current directory. *IBM.*

cursor. A reference to an element at a specific position in a data structure.

Customer Information Control System. See *CICS*.

c89 utility. A utility used to compile and bind an application program from the z/VM OpenExtensions shell.

D

data abstraction. A data type with a private representation and a public set of operations (functions or operators) which restrict access to that data type to that set of operations. The C++ language uses the concept of classes to implement data abstraction.

data definition (DD). (1) In the C and C++ languages, a definition that describes a data object, reserves storage for a data object, and can provide an initial value for a data object. A data definition appears outside a function or at the beginning of a block statement. *IBM.* (2) A program statement that describes the features of, specifies relationships of, or establishes context of, data. *ANSI/ISO.* (3) A statement that is

stored in the environment and that externally identifies a file and the attributes with which it should be opened.

data definition name. See *ddname*.

data definition statement. See *DD statement*.

data member. The smallest possible piece of complete data. Elements are composed of data members.

data object. (1) A storage area used to hold a value. (2) Anything that exists in storage and on which operations can be performed, such as files, programs, classes, or arrays. (3) In a program, an element of data structure, such as a file, array, or operand, that is needed for the execution of a program and that is named or otherwise specified by the allowable character set of the language in which a program is coded. *IBM.*

data set. Under z/OS, a named collection of related data records that is stored and retrieved by an assigned name.

data stream. A continuous stream of data elements being transmitted, or intended for transmission, in character or binary-digit form, using a defined format. *IBM.*

data structure. The internal data representation of an implementation.

data type. The properties and internal representation that characterize data.

Data Window Services (DWS). Services provided as part of the Callable Services Library that allow manipulation of data objects such as VSAM linear data sets and temporary data objects known as *TEMPSPACE*.

DBCS (double-byte character set). A set of characters in which each character is represented by 2 bytes. Languages such as Japanese, Chinese, and Korean, which contain more symbols than can be represented by 256 code points, require double-byte character sets.

Because each character requires 2 bytes, the typing, display, and printing of DBCS characters requires hardware and programs that support DBCS. *IBM.*

ddname (data definition name). (1) The logical name of a file within an application. The ddname provides the means for the logical file to be connected to the physical file. (2) The part of the data definition before the equal sign. It is the name used in a call to fopen or freopen to refer to the data definition stored in the environment.

DD statement (data definition statement). (1) In z/OS, serves as the connection between the logical name of a file and the physical name of the file. (2) A job control statement that defines a file to the operating system, and is a request to the operating system for the allocation of input/output resources.

dead code elimination. A process that eliminates code that exists for calculations that are not necessary. Code may be designated as dead by other optimization techniques.

dead store elimination. A process that eliminates unnecessary storage use in code. A store is deemed unnecessary if the value stored is never referenced again in the code.

decimal constant. (1) A numerical data type used in standard arithmetic operations. (2) A number containing any of the digits 0 through 9. *IBM.*

decimal overflow. A condition that occurs when one or more nonzero digits are lost because the destination field in a decimal operation is too short to contain the results.

declaration. (1) In the C and C++ languages, a description that makes an external object or function available to a function or a block statement. *IBM.* (2) Establishes the names and characteristics of data objects and functions used in a program.

declarator. Designates a data object or function declared. Initializations can be performed in a declarator.

default argument. An argument that is declared with a default value in a function prototype or declaration. If a call to the function omits this argument, the default value is used. Arguments with default values must be the trailing arguments in a function prototype argument list.

default clause. In the C or C++ languages, within a switch statement, the keyword default followed by a colon, and one or more statements. When the conditions of the specified case labels in the switch statement do not hold, the default clause is chosen. *IBM.*

default constructor. A constructor that takes no arguments, or, if it takes arguments, all its arguments have default values.

default initialization. The initial value assigned to a data object by the compiler if no initial value is specified by the programmer.

default locale. (1) The C locale, which is always used when no selection of locale is performed. (2) A system default locale, named by locale-related environmental variables.

define directive. A preprocessor directive that directs the preprocessor to replace an identifier or macro invocation with special code.

definition. (1) A data description that reserves storage and may provide an initial value. (2) A declaration that allocates storage, and may initialize a data object or specify the body of a function.

degree. The number of children of a node.

delete. (1) A C++ keyword that identifies a free storage deallocation operator. (2) A C++ operator used to destroy objects created by `new`.

demangling. The conversion of mangled names back to their original source code names. During C++ compilation, identifiers such as function and static class member names are mangled (encoded) with type and scoping information to ensure type-safe linkage. These mangled names appear in the object file and the final executable file. Demangling (decoding) converts these names back to their original names to make program debugging easier. See also *mangling*.

deque. A queue that can have elements added and removed at both ends. A double-ended queue.

dequeue. An operation that removes the first element of a queue.

dereference. In the C and C++ languages, the application of the unary operator * to a pointer to access the object the pointer points to. Also known as *indirection*.

derivation. In the C++ language, to derive a class, called a derived class, from an existing class, called a base class.

derived class. A class that inherits from a base class. All members of the base class become members of the derived class. You can add additional data members and member functions to the derived class. A derived class object can be manipulated as if it is a base class object. The derived class can override virtual functions of the base class.

descriptor. PL/I control block that holds information such as string lengths, array subscript bounds, and area sizes, and is passed from one PL/I routine to another during Run-Time.

destructor. A special member function that has the same name as its class, preceded by a tilde (~), and that ″cleans up″ after an object of that class, for example, freeing storage that was allocated when the object was created. A destructor has no arguments and no return type.

detach state attribute. An attribute associated with a thread attribute object. This attribute has two possible values:

0 Undetached. An undetached thread keeps its resources after termination of the thread.

1 Detached. A detached thread has its resources freed by the system after termination.

device. A computer peripheral or an object that appears to the application as such. *X/Open. ISO.1.*

difference. For two sets A and B, the difference (A-B) is the set of all elements in A but not in B. For bags, there is an additional rule for duplicates: If bag P contains an element *m* times and bag Q contains the same element *n* times, then, if *m*>n, the difference contains that element *m-n* times. If *m*≤*n*, the difference contains that element zero times.

digraph. A combination of two keystrokes used to represent unavailable characters in a C or C++ source program. Digraphs are read as tokens during the preprocessor phase.

directory. (1) In a hierarchical file system, a container for files or other directories. *IBM.* (2) The part of a partitioned data set that describes the members in the data set.

disabled signal. Synonym for *enabled signal*.

display. To direct the output to the user's terminal. If the output is not directed to the terminal, the results are undefined. *X/Open.*

DLL. See *dynamic link library*.

do statement. In the C and C++ compilers, a looping statement that contains the keyword `do`, followed by a statement (the action), the keyword `while`, and an expression in parentheses (the condition). *IBM.*

dot. The file name consisting of a single dot character (.). *X/Open. ISO.1.*

double-byte character set. See *DBCS*.

double-precision. Pertaining to the use of two computer words to represent a number in accordance with the required precision. *ISO-JTC1. ANSI/ISO.*

double-quote. The character ″, also known as *quotation mark*. *X/Open.*

This character is named <quotation-mark> in the portable character set.

doubleword. A contiguous sequence of bytes or characters that comprises two computer words and is capable of being addressed as a unit. *IBM.*

DSA (dynamic storage area). An area of storage obtained during the running of an application that consists of a register save area and an area for automatic data, such as program variables. DSAs are generally allocated within Language Environment-managed stack segments. DSAs are added to the stack when a routine is entered and

removed upon exit in a last in, first out (LIFO) manner. In Language Environment, a DSA is known as a stack frame.

dump. To copy data in a readable format from main or auxiliary storage onto an external medium such as tape, diskette, or printer. *IBM.*

dynamic. Pertaining to an operation that occurs at the time it is needed rather than at a predetermined or fixed time. *IBM.*

dynamic allocation. Assignment of system resources to a program when the program is executed rather than when it is loaded into main storage. *IBM.*

dynamic binding. The act of resolving references to external variables and functions at Run-Time. In C++, dynamic binding is supported by using virtual functions.

dynamic link library (DLL). A file containing executable code and data bound to a program at Run-Time. The code and data in a dynamic link library can be shared by several applications simultaneously. Compiling code with the DLL option does not mean that the produced executable will be a DLL. To create a DLL, use `#pragma export` or the `EXPORTALL` compiler option.

dynamic storage. Synonym for *automatic storage*.

dynamic storage area. See *DSA*.

E

EBCDIC. See *extended binary-coded decimal interchange code*.

effective group ID. An attribute of a process that is used in determining various permissions, including file access permissions. This value is subject to change during the process lifetime, as described in the exec family of functions and `setgid()`. *X/Open. ISO.1.*

effective user ID. (1) The user ID associated with the last authenticated user or the last `setuid()` program. It is equal to either the real or the saved user ID. (2) The current user ID, but not necessarily the user's login ID; for example, a user logged in under a login ID may change to another user's ID. The ID to which the user changes becomes the effective user ID until the user switches back to the original login ID. All discretionary access decisions are based on the effective user ID. *IBM.* (3) An attribute of a process that is used in determining various permissions, including file access permissions. This value is subject to change during the process lifetime, as described in *exec* and `setuid()`. *X/Open. ISO.1.*

elaborated type specifier. A specifier typically used in an incomplete class declaration to qualify types that are otherwise hidden.

element. The component of an array, subrange, enumeration, or set.

element equality. A relation that determines if two elements are equal.

element occurrence. A single instance of an element in a collection. In a unique collection, element occurrence is synonymous with element value.

element value. All the instances of an element with a particular value in a collection. In a nonunique collection, an element value may have more than one occurrence. In a unique collection, element value is synonymous with element occurrence.

else clause. The part of an if statement that contains the word else, followed by a statement. The else clause provides an action that is started when the if condition evaluates to a value of zero (false). *IBM.*

empty line. A line consisting of only a new-line character. *X/Open.*

empty string. (1) A string whose first byte is a null byte. Synonymous with null string. *X/Open.* (2) A character array whose first element is a null character. *ISO.1.*

enabled signal. The occurrence of an enabled signal results in the default system response or the execution of an established signal handler. If disabled, the occurrence of the signal is ignored.

encapsulation. Hiding the internal representation of data objects and implementation details of functions from the client program. This enables the end user to focus on the use of data objects and functions without having to know about their representation or implementation.

enclave. In Language Environment, an independent collection of routines, one of which is designated as the main routine. An enclave is roughly analogous to a program or run unit.

enqueue. (1) An operation that adds an element as the last element to a queue. (2) Request control of a serially reusable resource.

entry point. The address or label of the first instruction that is executed when a routine is entered for execution.

enumeration constant. In the C or C++ language, an identifier, with an associated integer value, defined in an enumerator. An enumeration constant may be used anywhere an integer constant is allowed. *IBM.*

enumeration data type. (1) In the Fortran, C, and C++ language, a data type that represents a set of values that a user defines. *IBM.* (2) A type that

represents integers and a set of enumeration constants. Each enumeration constant has an associated integer value.

enumeration tag. In the C and C++ language, the identifier that names an enumeration data type. *IBM.*

enumeration type. An enumeration type defines a set of enumeration constants. In the C++ language, an enumeration type is a distinct data type that is not an integral type.

enumerator. In the C and C++ language, an enumeration constant and its associated value. *IBM.*

equivalence class. (1) A grouping of characters that are considered equal for the purpose of collation; for example, many languages place an uppercase character in the same equivalence class as its lowercase form, but some languages distinguish between accented and unaccented character forms for the purpose of collation. *IBM.* (2) A set of collating elements with the same primary collation weight.

Elements in an equivalence class are typically elements that naturally group together, such as all accented letters based on the same base letter.

The collation order of elements within an equivalence class is determined by the weights assigned on any subsequent levels after the primary weight. *X/Open.*

escape sequence. (1) A representation of a character. An escape sequence contains the \ symbol followed by one of the characters: `a, b, f, n, r, t, v, ', ", x, \,` or followed by one or more octal or hexadecimal digits. (2) A sequence of characters that represent, for example, non-printing characters, or the exact code point value to be used to represent variant and nonvariant characters regardless of code page. (3) In the C and C++ language, an escape character followed by one or more characters. The escape character indicates that a different code, or a different coded character set, is used to interpret the characters that follow. Any member of the character set used at Run-Time can be represented using an escape sequence. (4) A character that is preceded by a backslash character and is interpreted to have a special meaning to the operating system. (5) A sequence sent to a terminal to perform actions such as moving the cursor, changing from normal to reverse video, and clearing the screen. Synonymous with multibyte control. *IBM.*

exception. (1) Any user, logic, or system error detected by a function that does not itself deal with the error but passes the error on to a handling routine (also called throwing the exception). (2) In programming languages, an abnormal situation that may arise during execution, that may cause a deviation from the normal execution sequence, and for which facilities exist in a programming language to define, raise, recognize, ignore, and handle it; for example, (ON-) condition in PL/I, exception in ADA. *ISO-JTC1.*

exception handler. (1) Exception handlers are catch blocks in C++ applications. Catch blocks catch exceptions when they are thrown from a function enclosed in a try block. Try blocks, catch blocks, and throw expressions are the constructs used to implement formal exception handling in C++ applications. (2) A set of routines used to detect deadlock conditions or to process abnormal condition processing. An exception handler allows the normal running of processes to be interrupted and resumed. *IBM.*

executable. A load module or program object which has yet to be loaded into memory for execution.

executable file. A regular file acceptable as a new process image file by the equivalent of the *exec* family of functions, and thus usable as one form of a utility. The standard utilities described as compilers can produce executable files, but other unspecified methods of producing executable files may also be provided. The internal format of an executable file is unspecified, but a conforming application cannot assume an executable file is a text file. *X/Open.*

executable program. A program that has been link-edited and therefore can be run in a processor. *IBM.*

extended binary-coded data interchange code (EBCDIC). A coded character set of 256 8-bit characters. *IBM.*

extended-precision. Pertaining to the use of more than two computer words to represent a floating point number in accordance with the required precision. In z/VM four computer words are used for an extended-precision number.

extension. (1) An element or function not included in the standard language. (2) File name extension.

external data definition. A description of a variable appearing outside a function. It causes the system to allocate storage for that variable and makes that variable accessible to all functions that follow the definition and are located in the same file as the definition. *IBM.*

extern storage class specifier. A specifier that enables the programmer to declare objects and functions that several source files can use.

F

feature test macro (FTM). A macro (`#define`) used to determine whether a particular set of features will be included from a header. *X/Open. ISO.1.*

FIFO special file. A type of file with the property that data written to such a file is read on a first-in-first-out basis. Other characteristics of FIFOs are described in `open()`, `read()`, `write()`, and `lseek()`. *X/Open. ISO.1.*

file access permissions. The standard file access control mechanism uses the file permission bits. The bits are set at the time of file creation by functions such as `open()`, `creat()`, `mkdir()`, and `mkfifo()` and can be changed by `chmod()`. The bits are read by `stat()` or `fstat()`. *X/Open.*

file descriptor. (1) A positive integer that the system uses instead of the file name to identify an open file. (2) A per-process unique, non-negative integer used to identify an open file for the purpose of file access. *ISO.1.*

The value of a file descriptor is from zero to {OPEN_MAX}--which is defined in <limits.h>. A process can have no more than {OPEN_MAX} file descriptors open simultaneously. File descriptors may also be used to implement directory streams. *X/Open.*

file mode. An object containing the *file mode bits* and file type of a file, as described in `<sys/stat.h>`. *X/Open.*

file mode bits. A file's file permission bits, set-user-ID-on-execution bit (S_ISUID) and set-group-ID-on-execution bit (S_ISGID). *X/Open.*

file permission bits. Information about a file that is used, along with other information, to determine if a process has read, write, or execute/search permission to a file. The bits are divided into three parts: owner, group, and other. Each part is used with the corresponding file class of process. These bits are contained in the file mode, as described in `<sys/stat.h>`. The detailed usage of the file permission bits is described in *file access permissions*. *X/Open. ISO.1.*

file scope. A name declared outside all blocks, classes, and function declarations has file scope and can be used after the point of declaration in a source file.

filter. A command whose operation consists of reading data from standard input or a list of input files and writing data to standard output. Typically, its function is to perform some transformation on the data stream. *X/Open.*

first element. The element visited first in an iteration over a collection. Each collection has its own definition for first element. For example, the first element of a sorted set is the element with the smallest value.

flat collection. A collection that has no hierarchical structure.

float constant. (1) A constant representing a nonintegral number. (2) A number containing a decimal point, an exponent, or both a decimal point and an exponent. The exponent contains an e or E, an optional sign (+ or -), and one or more digits (0 through 9). *IBM.*

for statement. A looping statement that contains the word *for* followed by a for-initializing-statement, an optional condition, a semicolon, and an optional expression, all enclosed in parentheses.

foreground process. (1) A process that must run to completion before another command is issued. The foreground process is in the foreground process group, which is the group that receives the signals generated by a terminal. *IBM.* (2) A process that is a member of a foreground process group. *X/Open. ISO.1.*

foreground process group. (1) The group that receives the signals generated by a terminal. *IBM.* (2) A process group whose member processes have certain privileges, denied to processes in background process groups, when accessing their controlling terminal. Each session that has established a connection with a controlling terminal has exactly one process group of the session as the foreground process group of that controlling terminal. *X/Open. ISO.1.*

foreground process group ID. The process group ID of the foreground process group. *X/Open. ISO.1.*

form-feed character. A character in the output stream that indicates that printing should start on the next page of an output device. The formfeed is the character designated by '\f' in the C and C++ language. If the formfeed is not the first character of an output line, the result is unspecified. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the movement to the next page. *X/Open.*

forward declaration. A declaration of a class or function made earlier in a compilation unit, so that the declared class or function can be used before it has been defined.

freestanding application. (1) An application that is created to run without the run-time environment or library with which it was developed. (2) A C/C++ application that does not use the services of the dynamic C/C++ run-time library or of the Language Environment. Under C support, this ability is a feature of the System Programming C support.

free store. Dynamically allocated memory. New and delete are used to allocate and deallocate free store.

friend class. A class in which all the member functions are granted access to the private and protected members of another class. It is named in the declaration of another class and uses the keyword friend as a prefix to the class. For example, the following source code makes all the functions and data in class you friends of class me:

```
class me {
   friend class you;
     // ...
};
```

friend function. A function that is granted access to the private and protected parts of a class. It is named in the declaration of the other class with the prefix `friend`.

function. A named group of statements that can be called and evaluated and can return a value to the calling statement. *IBM.*

function call. An expression that moves the path of execution from the current function to a specified function and evaluates to the return value provided by the called function. A function call contains the name of the function to which control moves and a parenthesized list of values. *IBM.*

function declarator. The part of a function definition that names the function, provides additional information about the return value of the function, and lists the function parameters. *IBM.*

function definition. The complete description of a function. A function definition contains a sequence of specifiers (storage class, optional type, inline, virtual, optional friend), a function declarator, optional constructor-initializers, parameter declarations, optional const, and the block statement. Inline, virtual, friend, and const are not available with C.

function prototype. A function declaration that provides type information for each parameter. It is the first line of the function (header) followed by a semicolon (;). The declaration is required by the compiler at the time that the function is declared, so that the compiler can check the type.

function scope. Labels that are declared in a function have function scope and can be used anywhere in that function after their declaration.

function template. Provides a blueprint describing how a set of related individual functions can be constructed.

G

Generalization. Refers to a class, function, or static data member which derives its definition from a template. An instantiation of a template function would be a generalization.

Generalized Object File Format (GOFF). It is the strategic object module format for S/390. It extends the capabilities of object modules to contain more information than current object modules. It removes the limitations of the previous object module format and supports future enhancements. It is required for XPLINK.

generic class. Synonym for *class templates*.

global. Pertaining to information available to more than one program or subroutine. *IBM.*

global scope. Synonym for *file scope*.

global variable. A symbol defined in one program module that is used in other independently compiled program modules.

GMT (Greenwich Mean Time). The solar time at the meridian of Greenwich, formerly used as the prime basis of standard time throughout the world. GMT has been superseded by coordinated universal time (UTC).

graphic character. (1) A visual representation of a character, other than a control character, that is normally produced by writing, printing, or displaying. *ISO Draft.* (2) A character that can be displayed or printed. *IBM.*

Graphical Data Display Manager (GDDM). Pertaining to an IBM licensed program that provides a group of routines that allows pictures to be defined and displayed procedurally through function routines that correspond to graphic primitives. *IBM.*

Greenwich Mean Time. See *GMT*.

group ID. (1) A non-negative integer that is used to identify a group of system users. Each system user is a member of at least one group. When the identity of a group is associated with a process, a group ID value is referred to as a real group ID, an effective group ID, one of the supplementary group IDs or a saved set-group-ID. *X/Open.* (2) A non-negative integer, which can be contained in an object of type *gid_t*, that is used to identify a group of system users. *ISO.1.*

H

halfword. A contiguous sequence of bytes or characters that constitutes half a computer word and can be addressed as a unit. *IBM.*

hash function. A function that determines which category, or bucket, to put an element in. A hash function is needed when implementing a hash table.

hash table. (1) A data structure that divides all elements into (preferably) equal-sized categories, or buckets, to allow quick access to the elements. The hash function determines which bucket an element belongs in. (2) A table of information that is accessed by way of a shortened search key (that hash value). Using a hash table minimizes average search time.

header file. A text file that contains declarations used by a group of functions, programs, or users.

heap. An unordered flat collection that allows duplicate elements.

heap storage. An area of storage used for allocation of storage whose lifetime is not related to the execution of the current routine. The heap consists of the initial heap segment and zero or more increments.

hexadecimal constant. A constant, usually starting with special characters, that contains only hexadecimal digits. Three examples for the hexadecimal constant with value 0 would be '\x00', '0x0', or '0X00'.

High Level Assembler. An IBM licensed program. Translates symbolic assembler language into binary machine language.

hiperspace memory file. An IBM file used under z/OS to deal with memory files as large as 2 gigabytes. *IBM.*

HLASM. See *High Level Assembler*.

hooks. Instructions inserted into a program by a compiler at compile-time. Using hooks, you can set break-points to instruct the Debug Tool to gain control of the program at selected points during its execution.

hybrid code. Program statements that have not been internationalized with respect to code page, especially where data constants contain variant characters. Such statements can be found in applications written in older implementations of MVS, which required syntax statements to be written using code page IBM-1047 exclusively. Such applications cannot be converted from one code page to another using `iconv()`.

I

I18N. Abbreviation for *internationalization*.

identifier. (1) One or more characters used to identify or name a data element and possibly to indicate certain properties of that data element. *ANSI/ISO.* (2) In programming languages, a token that names a data object such as a variable, an array, a record, a subprogram, or a function. *ANSI/ISO.* (3) A sequence of letters, digits, and underscores used to identify a data object or function. *IBM.*

if statement. A conditional statement that contains the keyword if, followed by an expression in parentheses (the condition), a statement (the action), and an optional else clause (the alternative action). *IBM.*

ILC (interlanguage call). A function call made by one language to a function coded in another language. Interlanguage calls are used to communicate between programs written in different languages.

ILC (interlanguage communication). The ability of routines written in different programming languages to communicate. ILC support enables the application writer to readily build applications from component routines written in a variety of languages.

implementation-defined behavior. Application behavior that is not defined by the standards. The implementing compiler and library defines this behavior when a program contains correct program constructs or uses correct data. Programs that rely on implementation-defined behavior may behave differently on different C or C++ implementations. Refer to the z/OS C/C++ books that are listed in "C/C++ for z/VM and related publications" on page 5 for information about implementation-defined behavior in the z/OS and z/VM C/C++ environment. Contrast with *unspecified behavior* and *undefined behavior*.

IMS (Information Management System). Pertaining to an IBM database/data communication (DB/DC) system that can manage complex databases and networks. *IBM.*

include directive. A preprocessor directive that causes the preprocessor to replace the statement with the contents of a specified file.

include file. See *header file*.

include statement. In C/C++, a preprocessor statement that causes the preprocessor to replace the statement with the contents of a specified file. *IBM.*

incomplete class declaration. A class declaration that does not define any members of a class. Until a class is fully declared, or defined, you can only use the class name where the size of the class is not required. Typically an incomplete class declaration is used as a forward declaration.

incomplete type. A type that has no value or meaning when it is first declared. There are three incomplete types: void, arrays of unknown size and structures and unions of unspecified content. A void type can never be completed. Arrays of unknown size and structures or unions of unspecified content can be completed in further declarations.

indirection. (1) A mechanism for connecting objects by storing, in one object, a reference to another object. (2) In the C and C++ languages, the application of the unary operator * to a pointer to access the object to which the pointer points.

indirection class. Synonym for *reference class*.

induction variable. It is a controlling variable of a loop.

inheritance. A technique that allows the use of an existing class as the base for creating other classes.

initial heap. The C/C++ heap controlled by the HEAP run-time option and designated by a `heap_id` of 0. The initial heap contains dynamically allocated user data.

initializer. An expression used to initialize data objects. The C++ language, supports the following types of initializers:

- An expression followed by an assignment operator that is used to initialize fundamental data type objects or class objects that contain copy constructors.
- A parenthesized expression list that is used to initialize base classes and members that use constructors.

Both the C and C++ languages support an expression enclosed in braces ({ }), that is used to initialize aggregates.

inlined function. A function whose actual code replaces a function call. A function that is both declared and defined in a class definition is an example of an inline function. Another example is one which you explicitly declared inline by using the keyword inline. Both member and non-member functions can be inlined.

input stream. A sequence of control statements and data submitted to a system from an input unit. Synonymous with input job stream, job input stream. *IBM.*

instance. An object-oriented programming term synonymous with object. An instance is a particular instantiation of a data type. It is simply a region of storage that contains a value or group of values. For example, if a class `box` is previously defined, two instances of a class `box` could be instantiated with the declaration: `box box1, box2;`

instantiate. To create or generate a particular instance or object of a data type. For example, an instance `box1` of class `box` could be instantiated with the declaration: `box box1;`

instruction. A program statement that specifies an operation to be performed by the computer, along with the values or locations of operands. This statement represents the programmer's request to the processor to perform a specific operation.

instruction scheduling. An optimization technique that reorders instructions in code to minimize execution time.

integer constant. A decimal, octal, or hexadecimal constant.

integral object. A character object, an object having an enumeration type, an object having variations of the type `int`, or an object that is a bit field.

Interactive System Productivity Facility. See *ISPF*.

interlanguage call. See *ILC (interlanguage call)*.

interlanguage communication. See *ILC (interlanguage communication)*.

internationalization. The capability of a computer program to adapt to the requirements of different native languages, local customs, and coded character sets. *X/Open.*

Synonymous with *I18N*.

interoperability. The capability to communicate, execute programs, or transfer data among various functional units in a way that requires the user to have little or no knowledge of the unique characteristics of those units.

Interprocedural Analysis. See *IPA*.

interprocess communication. (1) The exchange of information between processes or threads through semaphores, queues, and shared memory. (2) The process by which programs communicate data to each other to synchronize their activities. Semaphores, signals, and internal message queues are common methods of inter-process communication.

I/O Stream library. A class library that provides the facilities to deal with many varieties of input and output.

IPA (Interprocedural Analysis). A process for performing optimizations across compilation units.

ISO (International Standards Organization). An organization of national standards bodies from various countries established to promote development of standards to facilitate international exchange of goods and services, and develop cooperation in intellectual, scientific, technological, and economic activity. *IBM.*

ISPF (Interactive System Productivity Facility). An IBM licensed program that serves as a full-screen editor and dialogue manager. Used for writing application programs, it provides a means of generating standard screen panels and interactive dialogues between the application programmer and terminal user. (ISPF)

iteration. The process of repeatedly applying a function to a series of elements in a collection until some condition is satisfied.

J

JCL (job control language). A control language used to identify a job to an operating system and to describe the job's requirement. *IBM.*

job control. A facility that allows users to selectively stop (suspend) the execution of a process and continue (resume) their execution at a later point.

The user typically employs this facility via the interactive interface jointly supplied by the terminal I/O driver and a command interpreter. *X/Open. ISO.1.*

K

keyword. (1) A predefined word reserved for the C and C++ languages, that may not be used as an identifier. (2) A symbol that identifies a parameter in JCL.

kind attribute. An attribute for a mutex attribute object. This attribute's value determines whether the mutex can be locked once or more than once for a thread and whether state changes to the mutex will be reported to the debug interface.

L

L-name. An external name in an object module produced by compiling with the LONGNAME option.

label. An identifier within or attached to a set of data elements. *ISO Draft.*

Language Environment. Pertaining to an IBM software product that provides a common run-time environment and run-time services to applications compiled by Language Environment-conforming compilers.

last element. The element visited last in an iteration over a collection. Each collection has its own definition for last element. For example, the last element of a sorted set is the element with the largest value.

late binding. Allowing the system to determine the specific class of the object and invoke the appropriate function implementations at Run-Time. Late binding or dynamic binding hides the differences between a group of related classes from the application program.

leaves. Nodes without children. Synonymous with terminals.

lexically. Relating to the left-to-right order of units.

library. (1) A collection of functions, calls, subroutines, or other data. *IBM.* (2) A set of object modules that can be specified in a link command.

line. A sequence of zero or more non-new-line characters plus a terminating new-line character. *X/Open.*

link. To interconnect items of data or portions of one or more computer programs; for example, linking of object programs by a linkage editor to produce an executable file.

linkage editor. Synonym for linker. The linkage editor has been replaced by the *binder* for the MVS/ESA, OS/390, z/OS and z/VM CMS operating systems. See *binder*.

Linkage. Refers to the binding between a reference and a definition. A function has internal linkage if the function is defined inline as part of the class, is declared with the inline keyword, or is a non-member function declared with the static keyword. All other functions have external linkage.

linker. A computer program for creating load modules from one or more object modules by resolving cross references among the modules and, if necessary, adjusting addresses. *IBM.*

literal. (1) In programming languages, a lexical unit that directly represents a value; for example, 14 represents the integer fourteen, ″APRIL″ represents the string of characters APRIL, 3.0005E2 represents the number 300.05. *ISO-JTC1.* (2) A symbol or a quantity in a source program that is itself data, rather than a reference to data. *IBM.* (3) A character string whose value is given by the characters themselves; for example, the numeric literal 7 has the value 7, and the character literal CHARACTERS has the value CHARACTERS. *IBM.*

loader. A routine, commonly a computer program, that reads data into main storage. *ANSI/ISO.*

load module. All or part of a computer program in a form suitable for loading into main storage for execution. A load module is usually the output of a linkage editor. *ISO Draft.*

local. (1) In programming languages, pertaining to the relationship between a language object and a block such that the language object has a scope contained in that block. *ISO-JTC1.* (2) Pertaining to that which is defined and used only in one subdivision of a computer program. *ANSI/ISO.*

local customs. The conventions of a geographical area or territory for such things as date, time, and currency formats. *X/Open.*

locale. The definition of the subset of a user's environment that depends on language and cultural conventions. *X/Open.*

localization. The process of establishing information within a computer system specific to the operation of particular native languages, local customs, and coded character sets. *X/Open.*

local scope. A name declared in a block has scope within the block, and can therefore only be used in that block.

Long name. An external name C++ name in an object module, or and external name in an object module created by the C compiler when the `LONGNAME` option is used. Long names are up to 1024 characters long and may contain both upper-case and lower-case characters.

lvalue. An expression that represents a data object that can be both examined and altered.

M

macro. An identifier followed by arguments (may be a parenthesized list of arguments) that the preprocessor replaces with the replacement code located in a preprocessor `#define` directive.

macro call. Synonym for *macro*.

macro instruction. Synonym for *macro*.

main function. An external function with the identifier main that is the first user function--aside from exit routines and C++ static object constructors--to get control when program execution begins. Each C and C++ program must have exactly one function named main.

makefile. A text file containing a list of your application's parts. The make utility uses makefiles to maintain application parts and dependencies.

make utility. Maintains all of the parts and dependencies for your application. The make utility uses a makefile to keep the parts of your program synchronized. If one part of your application changes, the make utility updates all other files that depend on the changed part. This utility is available under the z/VM OpenExtensions shell and by default, uses the c89 utility to recompile and bind your application.

mangling. The encoding during compilation of identifiers such as function and variable names to include type and scope information. These mangled names ensure type-safe linkage. See also *demangling*.

manipulator. A value that can be inserted into streams or extracted from streams to affect or query the behavior of the stream.

mask. A pattern of characters that controls the keeping, deleting, or testing of portions of another pattern of characters. *ISO-JTC1. ANSI/ISO.*

member. A data object or function in a structure, union, or class. Members can also be classes, enumerations, bit fields, and type names.

member function. (1) An operator or function that is declared as a member of a class. A member function has access to the private and protected data members and member functions of objects of its class. Member functions are also called methods. (2) A function that performs operations on a class.

method. In the C++ language, a synonym for *member function*.

method file. (1) A file that allows users to indicate to the localedef utility where to look for user-provided methods for processing user-designed codepages. (2) For ASCII locales, a file that defines the method functions to be used by C Run-Time locale-sensitive interfaces. A method file also identifies where the method functions can be found. IBM supplies several method files used to create its standard set of ASCII locales. Other method files can be created to support customized or user-created codepages. Such customized method files replace IBM-supplied charmap method functions with user-written functions.

migrate. To move to a changed operating environment, usually to a new release or version of a system. *IBM.*

mode. A collection of attributes that specifies a file's type and its access permissions. *X/Open. ISO.1.*

module. A program unit that usually performs a particular function or related functions, and that is distinct and identifiable with respect to compiling, combining with other units, and loading.

multibyte character. A mixture of single-byte characters from a single-byte character set and double-byte characters from a double-byte character set.

multicharacter collating element. A sequence of two or more characters that collate as an entity. For example, in some coded character sets, an accented character is represented by a non-spacing accent, followed by the letter. Other examples are the Spanish elements *ch* and *ll*. *X/Open.*

multiple inheritance. An object-oriented programming technique implemented in the C++ language through derivation, in which the derived class inherits members from more than one base class.

multitasking. A mode of operation that allows concurrent performance, or interleaved execution of two or more tasks. *ISO-JTC1. ANSI/ISO.*

mutex. A flag used by a semaphore to protect shared resources. The mutex is locked and unlocked by threads in a program. A mutex can only be locked by one thread at a time and can only be unlocked by the same thread that locked it. The current owner of a mutex is the thread that it is currently locked by. An unlocked mutex has no current owner.

mutex attribute object. Allows the user to manage the characteristics of mutexes in their application by defining a set of values to be used for the mutex during its creation. A mutex attribute object allows the user to create many mutexes with the same set of characteristics without redefining the same set of characteristics for each mutex created.

mutex object. Used to identify a mutex.

N

namespace. A category used to group similar types of identifiers.

named pipe. A FIFO file. Named pipes allow transfer of data between processes in a FIFO manner and synchronization of process execution. Allows processes to communicate even though they do not know what processes are on the other end of the pipe.

natural reentrancy. A program that contains no writable static and requires no additional processing to make it reentrant is considered naturally reentrant.

nested class. A class defined within the scope of another class.

nested enclave. A new enclave created by an existing enclave. The nested enclave that is created must be a new main routine within the process. See also child enclave and parent enclave.

newline character. A character that in the output stream indicates that printing should start at the beginning of the next line. The newline character is designated by '\n' in the C and C++ language. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the movement to the next line. *X/Open.*

nickname. Synonym for alias.

non-printing character. See *control character*.

NULL. In C, a pointer that does not point to a data object. *IBM.*

null character (NUL). The ASCII or EBCDIC character '\0' with the hex value `00`, all bits turned off. It is used to represent the absence of a printed or displayed character. This character is named <NUL> in the portable character set.

null pointer. The value that is obtained by converting the number 0 into a pointer; for example, `(void *) 0`. The C and C++ languages guarantee that this value will not match that of any legitimate pointer, so it is used by many functions that return pointers to indicate an error. *X/Open.*

null statement. A C or C++ statement that consists solely of a semicolon.

null string. (1) A string whose first byte is a null byte. Synonymous with *empty string*. *X/Open.* (2) A character array whose first element is a null character. *ISO.1.*

null value. A parameter position for which no value is specified. *IBM.*

null wide-character code. A wide-character code with all bits set to zero. *X/Open.*

number sign. The character #, also known as *pound sign* and *hash sign*. This character is named <number-sign> in the portable character set.

O

object. (1) A region of storage. An object is created when a variable is defined. An object is destroyed when it goes out of scope. (See also *instance*.) (2) In object-oriented design or programming, an abstraction consisting of data and the operations associated with that data. See also *class*. *IBM.* (3) An instance of a class.

object code. Machine-executable instructions, usually generated by a compiler from source code written in a higher level language (such as the C++ language). For programs that must be linked, object code consists of relocatable machine code.

object module. (1) All or part of an object program sufficiently complete for linking. Assemblers and compilers usually produce object modules. *ISO Draft.* (2) A set of instructions in machine language produced by a compiler from a source program. *IBM.*

object-oriented programming. A programming approach based on the concepts of data abstraction and inheritance. Unlike procedural programming techniques, object-oriented programming concentrates not on how something is accomplished, but on what data objects comprise the problem and how they are manipulated.

octal constant. The digit 0 (zero) followed by any digits 0 through 7.

open file. A file that is currently associated with a file descriptor. *X/Open. ISO.1.*

z/VM OpenExtensions. Pertaining to z/VM OpenExtensions, a part of the z/VM operating system. It consists of a POSIX system Application Programming Interface for the C language, a shell and utilities, and a dbx debugger, all in conformance to IEEE POSIX standards (ISO 9945-1: 1990/IEEE POSIX 1003.1-1990, IEEE POSIX 1003.1a, IEEE POSIX 1003.2, and IEEE POSIX 1003.4a).

operand. An entity on which an operation is performed. *ISO-JTC1. ANSI/ISO.*

operating system (OS). Software that controls functions such as resource allocation, scheduling, input/output control, and data management.

operator function. An overloaded operator that is either a member of a class or that takes at least one argument that is a class type or a reference to a class type.

operator precedence. In programming languages, an order relation defining the sequence of the application of operators within an expression. *ISO-JTC1.*

orientation of a stream. After application of an input or output function to a stream, it becomes either byte-oriented or wide-oriented. A byte-oriented stream is a stream that had a byte input or output function applied to it when it had no orientation. A wide-oriented stream is a stream that had a wide character input or output function applied to it when it had no orientation. A stream has no orientation when it has been associated with an external file but has not had any operations performed on it.

overflow. (1) A condition that occurs when a portion of the result of an operation exceeds the capacity of the intended unit of storage. (2) That portion of an operation that exceeds the capacity of the intended unit of storage. *IBM.*

overlay. The technique of repeatedly using the same areas of internal storage during different stages of a program. *ANSI/ISO.* Unions are used to accomplish this in C and C++.

overloading. An object-oriented programming technique that allows you to redefine functions and most standard C++ operators when the functions and operators are used with class types.

P

pack. To store data in a compact form in such a way that the original form can be recovered.

parameter. (1) In the C and C++ languages, an object declared as part of a function declaration or definition that acquires a value on entry to the function, or an identifier following the macro name in a function-like macro definition. *X/Open.* (2) Data passed between programs or procedures. *IBM.*

parameter declaration. A description of a value that a function receives. A parameter declaration determines the storage class and the data type of the value.

parent enclave. The enclave that issues a call to system services or language constructs to create a nested or child enclave. See also *child enclave* and *nested enclave*.

parent process. (1) The program that originates the creation of other processes by means of spawn or exec function calls. See also *child process*. (2) A process that creates other processes.

parent process ID. (1) An attribute of a new process identifying the parent of the process. The parent process ID of a process is the process ID of its creator, for the lifetime of the creator. After the creator's lifetime has ended, the parent process ID is the process ID of an implementation-dependent system process. *X/Open.* (2) An attribute of a new process after it is created by a currently active process. *ISO.1.*

partitioned concatenation. Specifying multiple PDSs or PDSEs under one ddname. The concatenated data sets act as one big PDS or PDSE and access can be made to any member with a unique name. An attempted access to a member whose name occurs more than once in the concatenated data sets, returns the first member with that name found in the entire concatenation.

partitioned data set (PDS). A data set in direct access storage that is divided into partitions, called members, each of which can contain a program, part of a program, or data. *IBM.*

path name. (1) A string that is used to identify a file. A path name consists of, at most, {PATH_MAX} bytes, including the terminating null character. It has an optional beginning slash, followed by zero or more file names separated by slashes. If the path name refers to a directory, it may also have one or more trailing slashes. Multiple successive slashes are treated as one slash. A path name that begins with two successive slashes may be interpreted in an implementation-dependent manner, although more than two leading slashes are treated as a single slash. The interpretation of the path name is described in *path name resolution*. *ISO.1.* (2) A file name specifying all directories leading to the file.

path name resolution. Path name resolution is performed for a process to resolve a path name to a particular file in a file hierarchy. There may be multiple path names that resolve to the same file. *X/Open.*

pattern. A sequence of characters used either with regular expression notation or for path name expansion, as a means of selecting various characters strings or path names, respectively. The syntaxes of the two patterns are similar, but not identical. *X/Open.*

period. The character (.). The term *period* is contrasted against *dot*, which is used to describe a specific directory entry. This character is named <period> in the portable character set.

permissions. Codes that determine how a file can be used by any users who work on the system. See also *file access permissions*. *IBM.*

persistent environment. A program can explicitly establish a persistent environment, direct functions to it, and explicitly terminate it.

pointer. In the C and C++ languages, a variable that holds the address of a data object or a function. *IBM.*

pointer class. A class that implements pointers.

pointer to member. An operator used to access the address of non-static members of a class.

polymorphism. The technique of taking an abstract view of an object or function and using any concrete objects or arguments that are derived from this abstract view.

portable character set. The set of characters specified in POSIX 1003.2, section 2.4:

```
<NUL>
<alert>
<backspace>
<tab>
<newline>
<vertical-tab>
<form-feed>
<carriage-return>
<space>
<exclamation-mark>      !
<quotation-mark>        "
<number-sign>           #
<dollar-sign>           $
<percent-sign>          %
<ampersand>             &
<apostrophe>            '
<left-parenthesis>      (
<right-parenthesis>     )
<asterisk>              *
<plus-sign>             +
<comma>                 ,
<hyphen>                -
<hyphen-minus>          -
<period>                .
<slash>                 /
<zero>                  0
<one>                   1
<two>                   2
<three>                 3
<four>                  4
<five>                  5
<six>                   6
<seven>                 7
<eight>                 8
<nine>                  9
<colon>                 :
<semicolon>             ;
<less-than-sign>        <
<equals-sign>           =
<greater-than-sign>     >
<question-mark>         ?
<commercial-at>         @

<A>                     A
<B>                     B
<C>                     C
<D>                     D
<E>                     E
<F>                     F
<G>                     G
<H>                     H
<I>                     I
<J>                     J
<K>                     K
<L>                     L
<M>                     M
<N>                     N
<O>                     O
<P>                     P
<Q>                     Q
<R>                     R
<S>                     S
<T>                     T
<U>                     U
<V>                     V
<W>                     W
<X>                     X
<Y>                     Y
<Z>                     Z

<left-square-bracket>   [
<backslash>             \
<reverse-solidus>       \
<right-square-bracket>  ]
<circumflex>            ^
<circumflex-accent>     ^
<underscore>            _
<low-line>              _
<grave-accent>          `
<a>                     a
<b>                     b
<c>                     c
<d>                     d
<e>                     e
<f>                     f
<g>                     g
<h>                     h
<i>                     i
<j>                     j
<k>                     k
<l>                     l
<m>                     m
<n>                     n
<o>                     o
<p>                     p
<q>                     q
<r>                     r
<s>                     s
<t>                     t
<u>                     u
<v>                     v
<w>                     w
<x>                     x
<y>                     y
<z>                     z

<left-brace>            {
<left-curly-bracket>    {
<vertical-line>         |
<right-brace>           }
<right-curly-bracket>   }
<tilde>                 ~
```

portable file name character set. The set of characters from which portable file names are constructed. For a file name to be portable across implementations conforming to the ISO POSIX-1 standard and to ISO/IEC 9945, it must consists only of the following characters:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9 . _ -
```

The last three characters are the period, underscore, and hyphen characters, respectively. The hyphen must not be used as the first character of a portable file name. Upper- and lower-case letters retain their unique identities between conforming implementations. In the case of a portable path name, the slash character may also be used. *X/Open. ISO.1.*

portability. The ability of a programming language to compile successfully on different operating systems without requiring changes to the source code.

positional parameter. A parameter that must appear in a specified location relative to other positional parameters. *IBM.*

precedence. The priority system for grouping different types of operators with their operands.

predefined macros. Frequently used routines provided by an application or language for the programmer.

preinitialization. A process by which an environment or library is initialized once and can then be used repeatedly to avoid the inefficiency of initializing the environment or library each time it is needed.

prelinker. A utility provided with Language Environment that you can use to process application programs that require DLL support, or contain either constructed reentrancy or external symbol names that are longer than 8 characters. You require the prelinker, or its equivalent function which is provided by the binder, to process all C++ applications, or C applications that are compiled with the RENT, DLL, LONGNAME or IPA options. The prelinker is superseded by the binder. See also *binder*.

preprocessor. A phase of the compiler that examines the source program for preprocessor statements that are then executed, resulting in the alteration of the source program.

preprocessor statement. In the C and C++ languages, a statement that begins with the symbol # and is interpreted by the preprocessor during compilation. *IBM.*

primary expression. (1) An identifier, parenthesized expression, function call, array element specification, structure member specification, or union member specification. *IBM.* (2) Literals, names, and names qualified by the :: (scope resolution) operator.

printable character. One of the characters included in the print character classification of the LC_CTYPE category in the current locale. *X/Open.*

private. Pertaining to a class member that is only accessible to member functions and friends of that class.

privilege. See *file access permissions*.

process. (1) An instance of an executing application and the resources it uses. (2) An address space and single thread of control that executes within that address space, and its required system resources. A process is created by another process issuing the `fork()` function. The process that issues the `fork()` function is known as the parent process, and the new process created by the `fork()` function is known as the child process. *X/Open. ISO.1.*

process group. A collection of processes that permits the signaling of related processes. Each process in the system is a member of a process group that is identified by the process group ID. A newly created process joins the process group of its creator. *IBM. X/Open. ISO.1.*

process group ID. The unique identifier representing a process group during its lifetime. A process group ID is a positive integer. (Under ISO only, it is a positive integer *that can be contained in a pid_t*.) A process group ID will not be reused by the system until the process group lifetime ends. *X/Open. ISO.1.*

process group lifetime. A period of time that begins when a process group is created and ends when the last remaining process in the group leaves the group, because either it is the end of the last process' lifetime or the last remaining process is calling the `setsid()` or `setpgid()` functions. *X/Open. ISO.1.*

process ID. The unique identifier representing a process. A process ID is a positive integer. (Under ISO only, it is a positive integer *that can be contained in a pid_t*.) A process ID will not be reused by the system until the process lifetime ends. In addition, if there exists a process group whose process group ID is equal to that process ID, the process ID will not be reused by the system until the process group lifetime ends. A process that is not a system process will not have a process ID of 1. *X/Open. ISO.1.*

process lifetime. The period of time that begins when a process is created and ends when the process ID is returned to the system. After a process is created with a `fork()` function, it is considered active. Its thread of control and address space exist until it terminates. It then enters an inactive state where certain resources may be returned to the system, although some resources, such as the process ID, are still in use. When another process executes a `wait()` or `waitpid()` function for an inactive process, the remaining resources are returned to the system. The last resource to be returned to the system is the process ID. At this time, the lifetime of the process ends. *X/Open. ISO.1.*

program object. All or part of a computer program in a from suitable for loading into main storage for execution. A program object is the output of the Binder and is a newer more flexible format (e.g. longer external names) than a load module.

protected. Pertaining to a class member that is only accessible to member functions and friends of that class, or to member functions and friends of classes derived from that class.

prototype. A function declaration or definition that includes both the return type of the function and the types of its parameters. See *function prototype*.

public. Pertaining to a class member that is accessible to all functions.

pure virtual function. A virtual function that has a function definition of `= 0;`. See also *abstract classes*.

Q

qualified class name. Any class name or class name qualified with one or more :: (scope resolution) operators.

qualified name. Used to qualify a non-class type name such as a member by its class name.

qualified type name. Used to reduce complex class name syntax by using typedefs to represent qualified class names.

Query Management Facility (QMF). Pertaining to an IBM query and report writing facility that enables a variety of tasks such as data entry, query building, administration, and report analysis. *IBM.*

queue. A sequence with restricted access in which elements can only be added at the back end (or bottom) and removed from the front end (or top). A queue is characterized by first-in, first-out behavior and chronological order.

quotation marks. The characters ″ and ’, also known as *double-quote* and *single-quote* respectively. *X/Open.*

R

radix character. The character that separates the integer part of a number from the fractional part. *X/Open.*

real group ID. The attribute of a process that, at the time of process creating, identifies the group of the user who created the process. This value is subject to change during the process lifetime, as describe in `setgid()`. *X/Open. ISO.1.*

real user ID. The attribute of a process that, at the time of process creation, identifies the user who created the process. This value is subject to change during the process lifetime, as described in `setuid()`. *X/Open. ISO.1.*

reason code. A code that identifies the reason for a detected error. *IBM.*

reassociation. An optimization technique that rearranges the sequence of calculations in a subscript expression producing more candidates for common expression elimination.

redirection. In the shell, a method of associating files with the input or output of commands. *X/Open.*

reentrant. The attribute of a program or routine that allows the same copy of a program or routine to be used concurrently by two or more tasks.

reference class. A class that links a concrete class to an abstract class. Reference classes make polymorphism possible with the Collection Classes. Synonymous with *indirection class*.

refresh. To ensure that the information on the user’s terminal screen is up-to-date. *X/Open.*

register storage class specifier. A specifier that indicates to the compiler within a block scope data definition, or a parameter declaration, that the object being described will be heavily used.

register variable. A variable defined with the register storage class specifier. Register variables have automatic storage.

regular expression. (1) A mechanism to select specific strings from a set of character strings. (2) A set of characters, meta-characters, and operators that define a string or group of strings in a search pattern. (3) A string containing wildcard characters and operations that define a set of one or more possible strings.

regular file. A file that is a randomly accessible sequence of bytes, with no further structure imposed by the system. *X/Open. ISO.1.*

relation. An unordered flat collection class that uses keys, allows for duplicate elements, and has element equality.

relative path name. The name of a directory or file expressed as a sequence of directories followed by a file name, beginning from the current directory. See *path name resolution*. *IBM.*

reserved word. (1) In programming languages, a keyword that may not be used as an identifier. *ISO-JTC1.* (2) A word used in a source program to describe an action to be taken by the program or compiler. It must not appear in the program as a user-defined name or a system name. *IBM.*

RMODE (residency mode). A program attribute that refers to where a module is prepared to run. RMODE can be 24 or ANY. ANY refers to the fact that the module can be loaded either above or below the 16M line. RMODE 24 means the module expects to be loaded below the 16M line.

root. (1) A node that has no parent. All other nodes of a tree are descendants of the root. (2) In the AIX operating system, the user name for the system user with the most authority. *IBM.*

RTTI. Use the RTTI option to generate run-time type identification (RTTI) information for the typeid operator and the dynamic_cast operator.

run-time library. A compiled collection of functions whose members can be referred to by an application program during run-time execution. Typically used to refer to a dynamic library that is provided in object code, such that references to the library are resolved during the linking step. The run-time library itself is not statically bound into the application modules.

S

S-name. An external name in an object module produced by compiling with the NOLONGNAME option. Such a name is up to 8 characters long and single case.

saved set-group-ID. An attribute of a process that allows some flexibility in the assignment of the effective group ID attribute, as described in the `exec()` family of functions and `setgid()`. *X/Open. ISO.1.*

saved set-user-ID. An attribute of a process that allows some flexibility in the assignment of the effective user ID attribute, as described in `exec()` and `setuid()`. *X/Open. ISO.1.*

scalar. An arithmetic object, or a pointer to an object of any type.

scope. (1) That part of a source program in which a variable is visible. (2) That part of a source program in which an object is defined and recognized.

scope operator (::). An operator that defines the scope for the argument on the right. If the left argument is blank, the scope is global; if the left argument is a class name, the scope is within that class. Synonymous with *scope resolution operator*.

scope resolution operator (::). Synonym for *scope operator*.

semaphore. An object used by multi-threaded applications for signalling purposes and for controlling access to serially reusable resources. Processes can be locked to a resource with semaphores if the processes follow certain programming conventions.

sequence. A sequentially ordered flat collection.

sequential concatenation. Multiple sequential data sets or partitioned data-set members are treated as one long sequential data set. In the case of sequential data sets, you can access or update the data sets in order. In the case of partitioned data-set members, you can access or update the members in order. Repositioning is possible if all of the data sets in the concatenation support repositioning.

sequential data set. A data set whose records are organized on the basis of their successive physical positions, such as on magnetic tape. *IBM.*

session. A collection of process groups established for job control purposes. Each process group is a member of a session. A process is a member of the session of which its process group is a member. A newly created process joins the session of its creator. A process can alter its session membership; see `setsid()`. There can be multiple process groups in the same session. *X/Open. ISO.1.*

shell. A program that interprets sequences of text input as commands. It may operate on an input stream or it may interactively prompt and read commands from a terminal. *X/Open.*

This feature is provided as part of the z/VM OpenExtensions Shell and Utilities feature licensed program.

Short name. An external non-C++ name in an object module produced by compiling with the NOLONGNAME option. Such a name is up to 8 characters long and single case.

signal. (1) A condition that may or may not be reported during program execution. For example, `SIGFPE` is the signal used to represent erroneous arithmetic operations such as a division by zero. (2) A mechanism by which a process may be notified of, or affected by, an event occurring in the system. Examples of such events include hardware exceptions and specific actions by processes. The term *signal* is also used to refer to the event itself. *X/Open. ISO.1.* (3) A method of interprocess communication that simulates software interrupts. *IBM.*

signal handler. A function to be called when the signal is reported.

single-byte character set (SBCS). A set of characters in which each character is represented by a one-byte code. *IBM.*

single-precision. Pertaining to the use of one computer word to represent a number in accordance with the required precision. *ISO-JTC1. ANSI/ISO.*

single-quote. The character ', also known as *apostrophe*. This character is named <quotation-mark> in the portable character set.

slash. The character /, also known as *solidus*. This character is named <slash> in the portable character set.

socket. (1) A unique host identifier created by the concatenation of a port identifier with a transmission

control protocol/Internet protocol (TCP/IP) address. (2) A port identifier. (3) A 16-bit port-identifier. (4) A port on a specific host; a communications end point that is accessible though a protocol family's addressing mechanism. A socket is identified by a socket address. *IBM.*

sorted map. A sorted flat collection with key and element equality.

sorted relation. A sorted flat collection that uses keys, has element equality, and allows duplicate elements.

sorted set. A sorted flat collection with element equality.

source file. A file that contains source statements for such items as high-level language programs and data description specifications. *IBM.*

source module. A file that contains source statements for such items as high-level language programs and data description specifications. *IBM.*

source program. A set of instructions written in a programming language that must be translated to machine language before the program can be run. *IBM.*

space character. The character defined in the portable character set as <space>. The space character is a member of the space character class of the current locale, but represents the single character, and not all of the possible members of the class. *X/Open.*

spanned record. A logical record contained in more than one block. *IBM.*

specialization. A user-supplied definition which replaces a corresponding template instantiation.

specifiers. Used in declarations to indicate storage class, fundamental data type and other properties of the object or function being declared.

spill area. A storage area used to save the contents of registers. *IBM.*

SQL (Structured Query Language). A language designed to create, access, update and free data tables.

square brackets. The characters [(left bracket) and] (right bracket). Also see *brackets*.

stack frame. The physical representation of the activation of a routine. The stack frame is allocated and freed on a LIFO (last in, first out) basis. A stack is a collection of one or more stack segments consisting of an initial stack segment and zero or more increments.

stack storage. Synonym for *automatic storage*.

standard error. An output stream usually intended to be used for diagnostic messages. *X/Open.*

standard input. (1) An input stream usually intended to be used for primary data input. *X/Open.* (2) The primary source of data entered into a command. Standard input comes from the keyboard unless redirection or piping is used, in which case standard input can be from a file or the output from another command. *IBM.*

standard output. (1) An output stream usually intended to be used for primary data output. *X/Open.* (2) The primary destination of data coming from a command. Standard output goes to the display unless redirection or piping is used, in which case standard output can go to a file or to another command. *IBM.*

statement. An instruction that ends with the character ; (semicolon) or several instructions that are surrounded by the characters { and }.

static. A keyword used for defining the scope and linkage of variables and functions. For internal variables, the variable has block scope and retains its value between function calls. For external values, the variable has file scope and retains its value within the source file. For class variables, the variable is shared by all objects of the class and retains its value within the entire program.

static binding. The act of resolving references to external variables and functions before Run-Time.

storage class specifier. One of the terms used to specify a storage class, such as auto, register, static, or extern.

stream. (1) A continuous stream of data elements being transmitted, or intended for transmission, in character or binary-digit form, using a defined format. (2) A file access object that allows access to an ordered sequence of characters, as described by the ISO C standard. Such objects can be created by the `fdopen()` or `fopen()` functions, and are associated with a file descriptor. A stream provides the additional services of user-selectable buffering and formatted input and output. *X/Open.*

string. A contiguous sequence of bytes terminated by and including the first null byte. *X/Open.*

string constant. Zero or more characters enclosed in double quotation marks.

string literal. Zero or more characters enclosed in double quotation marks.

striped data set. A special data set organization that spreads a data set over a specified number of volumes so that I/O parallelism can be exploited. Record n in a striped data set is found on a volume separate from the volume containing record *n - p*, where *n* > p.

struct. An aggregate of elements having arbitrary types.

structure. A construct (a class data type) that contains an ordered group of data objects. Unlike an array, the data objects within a structure can have varied data types. A structure can be used in all places a class is used. The initial projection is public.

structure tag. The identifier that names a structure data type.

Structured Query Language. See *SQL*.

stub routine. A routine, within a run-time library, that contains the minimum lines of code required to locate a given routine at Run-Time.

subscript. One or more expressions, each enclosed in brackets, that follow an array name. A subscript refers to an element in an array.

subsystem. A secondary or subordinate system, usually capable of operating independently of or asynchronously with, a controlling system. *ISO Draft.*

subtree. A tree structure created by arbitrarily denoting a node to be the root node in a tree. A subtree is always part of a whole tree.

superset. Given two sets A and B, A is a superset of B if and only if all elements of B are also elements of A. That is, A is a superset of B if B is a subset of A.

support. In system development, to provide the necessary resources for the correct operation of a functional unit. *IBM.*

switch expression. The controlling expression of a switch statement.

switch statement. A C or C++ language statement that causes control to be transferred to one of several statements depending on the value of an expression.

system default. A default value defined in the system profile. *IBM.*

system process. (1) An implementation-dependent object, other than a process executing an application, that has a process ID. *X/Open.* (2) An object, other than a process executing an application, that is defined by the system, and has a process ID. *ISO.1.*

T

tab character. A character that in the output stream indicates that printing or displaying should start at the next horizontal tabulation position on the current line. The tab is the character designated by '\t' in the C language. If the current position is at or past the last defined horizontal tabulation position, the behavior is unspecified. It is unspecified whether the character is the exact sequence transmitted to an output device by the system to accomplish the tabulation. *X/Open.* This character is named <tab> in the portable character set.

task. (1) In a multiprogramming or multiprocessing environment, one or more sequences of instructions treated by a control program as an element of work to be accomplished by a computer. *ISO-JTC1. ANSI/ISO.* (2) A routine that is used to simulate the operation of programs. Tasks are said to be *nonpreemptive* because only a single task is executing at any one time. Tasks are said to be *lightweight* because less time and space are required to create a task than a true operating system process.

task library. A class library that provides the facilities to write programs that are made up of tasks.

template. A family of classes or functions with variable types.

template class. A class instance generated by a class template.

template function. A function generated by a function template.

template instantiation. The act of creating a new definition of a function, class, or member of a class from a template declaration and one or more template arguments.

terminals. Synonym for *leaves*.

text file. A file that contains characters organized into one or more lines. The lines must not contain NUL characters and none can exceed {LINE_MAX}--which is defined in limits.h--bytes in length, including the new-line character. The term *text file* does not prevent the inclusion of control or other unprintable characters (other than NUL). *X/Open.*

thread. The smallest unit of operation to be performed within a process. *IBM.*

throw expression. An argument to the C++ exception being thrown.

tilde. The character ~. This character is named <tilde> in the portable character set.

token. The smallest independent unit of meaning of a program as defined either by a parser or a lexical analyzer. A token can contain data, a language keyword, an identifier, or other parts of language syntax. *IBM.*

traceback. A section of a dump that provides information about the stack frame, the program unit address, the entry point of the routine, the statement number, and the status of the routines on the call-chain at the time the traceback was produced.

trap. An unprogrammed conditional jump to a specified address that is automatically activated by hardware. A recording is made of the location from which the jump occurred. *ISO-JTC1.*

trigraph sequence. An alternative spelling of some characters to allow the implementation of C in character sets that do not provide a sufficient number of non-alphabetic graphics. *ANSI/ISO.*

Before preprocessing, each trigraph sequence in a string or literal is replaced by the single character that it represents.

truncate. To shorten a value to a specified length.

try block. A block in which a known C++ exception is passed to a handler.

type. The description of the data and the operations that can be performed on or by the data. See also *data type*.

type conversion. Synonym for *boundary alignment*.

type definition. A definition of a name for a data type. *IBM.*

type specifier. Used to indicate the data type of an object or function being declared.

U

ultimate consumer. The target of data in an I/O operation. An ultimate consumer can be a file, a device, or an array of bytes in memory.

ultimate producer. The source of data in an I/O operation. An ultimate producer can be a file, a device, or an array of byes in memory.

unary expression. An expression that contains one operand. *IBM.*

undefined behavior. Action by the compiler and library when the program uses erroneous constructs or contains erroneous data. Permissible undefined behavior includes ignoring the situation completely with unpredictable results. It also includes behaving in a documented manner that is characteristic of the environment, during translation or program execution, with or without issuing a diagnostic message. It can also include terminating a translation or execution, while issuing a diagnostic message. Contrast with *unspecified behavior* and *implementation-defined behavior*.

underflow. (1) A condition that occurs when the result of an operation is less than the smallest possible nonzero number. (2) Synonym for arithmetic underflow, monadic operation. *IBM.*

union. (1) In the C or C++ language, a variable that can hold any one of several data types, but only one data type at a time. *IBM.* (2) For bags, there is an additional rule for duplicates: If bag P contains an element *m* times and bag Q contains the same element *n* times, then the union of P and Q contains that element *m+n* times.

union tag. The identifier that names a union data type.

unnamed pipe. A pipe that is accessible only by the process that created the pipe and its child processes. An unnamed pipe does not have to be opened before it can be used. It is a temporary file that lasts only until the last file descriptor that uses it is closed.

unique collection. A collection in which the value of an element only occurs once; that is, there are no duplicate elements.

unrecoverable error. An error for which recovery is impossible without use of recovery techniques external to the computer program or run.

unspecified behavior. Action by the compiler and library when the program uses correct constructs or data, for which the standards impose no specific requirements. Such action should not cause compiler or application failure. You should not, however, write any programs to rely on such behavior as they may not be portable to other systems. Contrast with *implementation-defined behavior* and *undefined behavior*.

user-defined data type. (1) A mathematical model that includes a structure for storing data and operations that can be performed on that data. Common abstract data types include sets, trees, and heaps. (2) See also *abstract data type*.

user ID. A nonnegative integer that is used to identify a system user. (Under ISO only, a nonnegative integer, which can be contained in an object of type *uid_t*.) When the identity of a user is associated with a process, a user ID value is referred to as a real user ID, an effective user ID, or (under ISO only, and there optionally) a saved set-user ID. *X/Open. ISO.1.*

user name. A string that is used to identify a user. *ISO.1.*

V

value numbering. An optimization technique that involves local constant propagation, local expression elimination, and folding several instructions into a single instruction.

variable. In programming languages, a language object that may take different values, one at a time. The values of a variable are usually restricted to a certain data type. *ISO-JTC1.*

variant character. A character whose hexadecimal value differs between different character sets. On EBCDIC systems, such as S/390, these 13 characters are an exception to the portability of the portable character set.

```
<left-square-bracket>  [
<right-square-bracket> ]
<left-brace>           {
<right-brace>          }
<backslash>            \
<circumflex>           ¬
<tilde>                ~
<exclamation-mark>     !
<number-sign>          #
<vertical-line>        |
<grave-accent>         `
<dollar-sign>          $
<commercial-at>        @
```

vertical-tab character. A character that in the output stream indicates that printing should start at the next vertical tabulation position. The vertical-tab is the character designated by '\v' in the C or C++ languages. If the current position is at or past the last defined vertical tabulation position, the behavior is unspecified. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the tabulation. *X/Open.* This character is named <vertical-tab> in the portable character set.

virtual address space. In virtual storage systems, the virtual storage assigned to a batched or terminal job, a system task, or a task initiated by a command.

virtual function. A function of a class that is declared with the keyword *virtual*. The implementation that is executed when you make a call to a virtual function depends on the type of the object for which it is called, which is determined at Run-Time.

Virtual Storage Access Method (VSAM). An access method for direct or sequential processing of fixed and variable length records on direct access devices. The records in a VSAM data set or file can be organized in logical sequence by a key field (key sequence), in the physical sequence in which they are written on the data set or file (entry-sequence), or by relative-record number.

visible. Visibility of identifiers is based on scoping rules and is independent of *access*.

volatile attribute. (1) In the C or C++ language, the keyword *volatile*, used in a definition, declaration, or cast. It causes the compiler to place the value of the data object in storage and to reload this value at each reference to the data object. *IBM.* (2) An attribute of a data object that indicates the object is changeable. Any expression referring to a volatile object is evaluated immediately (for example, assignments).

W

while statement. A looping statement that contains the keyword *while* followed by an expression in parentheses (the condition) and a statement (the action). *IBM.*

white space. (1) Space characters, tab characters, form-feed characters, and new-line characters. (2) A sequence of one or more characters that belong to the space character class as defined via the LC_CTYPE category in the current locale. In the POSIX locale, white space consists of one or more blank characters (space and tab characters), new-line characters, carriage-return characters, form-feed characters, and vertical-tab characters. *X/Open.*

wide-character. A character whose range of values can represent distinct codes for all members of the largest extended character set specified among the supporting locales.

wide-character code. An integral value corresponding to a single graphic symbol or control code. *X/Open.*

wide-character string. A contiguous sequence of wide-character codes terminated by and including the first null wide-character code. *X/Open.*

wide-oriented stream. See *orientation of a stream*.

word. A character string considered as a unit for a given purpose. In s/390, a word is 32 bits or 4 bytes.

working directory. Synonym for *current working directory*.

writable static area. See WSA.

write. (1) To output characters to a file, such as standard output or standard error. Unless otherwise stated, standard output is the default output destination for all uses of the term *write*. *X/Open.* (2) To make a permanent or transient recording of data in a storage device or on a data medium. *ISO-JTC1. ANSI/ISO.*

WSA (writable static area). An area of memory in the program that is modifyable during program execution. Typically, this area contains global variables and function and variable descriptors for DLLs.

X

XPLINK (Extra Performance Linkage). A new call linkage between functions that has the potential for a significant performance increase when used in an environment of frequent calls between small functions. XPLINK makes subroutine calls more efficient by removing nonessential instructions from the main path. When all functions are compiled with the XPLINK option, pointers can be used without restriction, which makes it easier to port new applications to z/VM CMS.

Bibliography

This bibliography lists the publications that provide information about your z/VM system. The z/VM library includes z/VM base publications, publications for additional facilities included with z/VM, and publications for z/VM optional features. For abstracts of z/VM publications and information about current editions and available publication formats, see z/VM: General Information.

IBM VM Internet Library

The latest editions of most z/VM publications are available as Adobe PDF files and IBM BookManager_ files from the IBM VM Internet Library:

`http://www.ibm.com/eserver/zseries/zvm/library/`

The IBM VM Internet Library also includes other information about z/VM, such as:

- Program directories
- Data areas and control blocks
- Monitor records

z/VM Base Publications

Evaluation

- *z/VM: General Information*, GC24-5991

Installation and Service

- *z/VM: Installation Guide*, GC24-5992
- *z/VM: Service Guide*, GC24-5993
- *z/VM: VMSES/E Introduction and Reference*, GC24-5994

Planning and Administration

- *z/VM: Planning and Administration*, SC24-5995
- *z/VM: CMS File Pool Planning, Administration, and Operation*, SC24-5949
- *z/VM: Migration Guide*, GC24-5996
- *z/VM: Running Guest Operating Systems*, SC24-5997
- *VM/ESA: Connectivity Planning, Administration, and Operation*, SC24-5756
- *z/VM: Group Control System*, SC24-5998
- *z/VM: Performance*, SC24-5999
- *z/VM: System Administration Facility*, SC24-6034

Customization

- *z/VM: CP Exit Customization*, SC24-5953

Operation

- *z/VM: System Operation*, SC24-6000
- *z/VM: Virtual Machine Operation*, SC24-6036

Application Programming

- *z/VM: CP Programming Services*, SC24-6001
- *z/VM: CMS Application Development Guide*, SC24-6002
- *z/VM: CMS Application Development Guide for Assembler*, SC24-6003
- *z/VM: CMS Callable Services Reference*, SC24-6004
- *z/VM: CMS Macros and Functions Reference*, SC24-6005
- *z/VM: CMS Application Multitasking*, SC24-5961
- *VM/ESA: REXX/VM Primer*, SC24-5598
- *z/VM: REXX/VM User's Guide*, SC24-5962
- *z/VM: REXX/VM Reference*, SC24-6035
- *z/VM: OpenExtensions POSIX Conformance Document*, GC24-5976
- *z/VM: OpenExtensions User's Guide*, SC24-5977
- *z/VM: OpenExtensions Command Reference*, SC24-6006
- *z/VM: OpenExtensions Advanced Application Programming Tools*, SC24-5979
- *z/VM: OpenExtensions Callable Services Reference*, SC24-6007
- *OS/390: DFSMS Program Management*, SC27-0806
- *z/VM: Program Management Binder for CMS*, SC24-5934
- *Debug Tool User's Guide and Reference*, SC09-2137
- *z/VM: Reusable Server Kernel Programmer's Guide and Reference*, SC24-5964
- *z/VM: Enterprise Systems Architecture/Extended Configuration Principles of Operation*, SC24-5965
- *External Security Interface (RACROUTE) Macro Reference for MVS and VM*, GC28-1366
- *VM/ESA: CPI Communications User's Guide*, SC24-5595
- *Common Programming Interface Communications Reference*, SC26-4399
- *Common Programming Interface Resource Recovery Reference*, SC31-6821
- *VM/ESA: Programmer's Guide to the Server-Requester Programming Interface for VM*, SC24-5455

End Use

- *z/VM: CP Command and Utility Reference*, SC24-6008
- *VM/ESA: CMS Primer*, SC24-5458
- *z/VM: CMS User's Guide*, SC24-6009
- *z/VM: CMS Command and Utility Reference*, SC24-6010
- *z/VM: CMS Pipelines User's Guide*, SC24-5970
- *z/VM: CMS Pipelines Reference*, SC24-5971
- *CMS/TSO Pipelines: Author's Edition*, SL26-0018
- *z/VM: XEDIT User's Guide*, SC24-5972
- *z/VM: XEDIT Command and Macro Reference*, SC24-5973
- *z/VM: Quick Reference*, SC24-6011

Diagnosis

- *z/VM: System Messages and Codes - CP*, GC24-6030
- *z/VM: System Messages and Codes - CMS*, GC24-6031
- *z/VM: System Messages and Codes - Other Components*, GC24-6032
- *z/VM: Diagnosis Guide*, GC24-6039
- *z/VM: VM Dump Tool*, GC24-5887
- *z/VM: Dump Viewing Facility*, GC24-5966

Publications for z/VM Additional Facilities

DFSMS/VM

- *VM/ESA: DFSMS/VM Function Level 221 Planning Guide*, GC35-0121
- *VM/ESA: DFSMS/VM Function Level 221 Installation and Customization*, SC26-4704
- *VM/ESA: DFSMS/VM Function Level 221 Storage Administration Guide and Reference*, SH35-0111
- *VM/ESA: DFSMS/VM Function Level 221 Removable Media Services User's Guide and Reference*, SC35-0141
- *VM/ESA: DFSMS/VM Function Level 221 Messages and Codes*, SC26-4707
- *VM/ESA: DFSMS/VM Function Level 221 Diagnosis Guide*, LY27-9589

Language Environment

- *Language Environment for OS/390 & VM: Concepts Guide*, GC28-1945
- *Language Environment for OS/390 & VM: Run-Time Migration Guide*, SC28-1944
- *Language Environment for OS/390 & VM: Programming Guide*, SC28-1939
- *Language Environment for OS/390 & VM: Programming Reference*, SC28-1940
- *Language Environment for OS/390 & VM: Writing Interlanguage Communication Applications*, SC28-1943
- *Language Environment for OS/390 & VM: Debugging Guide and Run-Time Messages*, SC28-1942
- *z/VM: Language Environment 1.8 C Run-Time Library Reference*, SC24-6038

OSA/SF

- *S/390: Planning for the S/390 Open Systems Adapter (OSA-1, OSA-2) Feature*, GC23-3870
- *zSeries 900: Planning for the Open Systems Adapter-2 Feature*, GA22-7477
- *VM/ESA: Open Systems Adapter Support Facility User's Guide for OSA-2*, SC28-1992
- *S/390: Open Systems Adapter-Express Customer's Guide and Reference*, SA22-7403
- *zSeries 900: Open Systems Adapter-Express Customer's Guide and Reference*, SA22-7476

TCP/IP for z/VM

- *z/VM: TCP/IP Level 420 Planning and Customization*, SC24-6019
- *z/VM: TCP/IP Level 420 User's Guide*, SC24-6020
- *z/VM: TCP/IP Level 420 Programmer's Reference*, SC24-6021
- *z/VM: TCP/IP Level 420 Messages and Codes*, GC24-6022
- *z/VM: TCP/IP Level 420 Diagnosis Guide*, GC24-6023

Publications for z/VM Optional Features

DirMaint

- *z/VM: Directory Maintenance Facility Function Level 410 Tailoring and Administration Guide*, SC24-6024
- *z/VM: Directory Maintenance Facility Function Level 410 Command Reference*, SC24-6025
- *z/VM: Directory Maintenance Facility Function Level 410 Messages*, GC24-6026

PRF

- *z/VM: Performance Reporting Facility Function Level 410*, SC24-6027

RTM

- *z/VM: RealTime Monitor Function Level 410*, SC24-6028

IBM VM Collection CD-ROM

The Online Library Omnibus Edition: VM Collection, SK2T-2067, contains all the IBM libraries that are available in BookManager format for current VM system products and current IBM licensed programs that run on VM. This CD-ROM also contains PDF versions of most z/VM publications and publications for some related IBM licensed programs.

Note: Only unlicensed publications are included.

Other Books You Might Need

The following list contains other IBM books that this publication references.

- *IBM z/Architecture Principles of Operation*, SA22-7832-00

z/OS C/C++ Publications

- *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

C for VM/ESA Publications

- *C for VM/ESA: Compiler and Run-Time Migration Guide*, SC09-2147
- *C for VM/ESA: Diagnosis Guide*, SC09-2149
- *C for VM/ESA: Language Reference*, SC09-2153
- *C for VM/ESA: User's Guide*, SC09-2152
- *C for VM/ESA: Programming Guide*, SC09-2151

Assembler Publications

- *IBM High Level Assembler Programmer's Guide*, SC26-4941
- *IBM High Level Assembler Language Reference*, SC26-4940

Cross System Product Publications

- *CSP/AD and High-Level Languages*, GG24-3362
- *CSP/AD Operation Development*, SH23-0502

COBOL for MVS & VM Publications

- *COBOL Programming Guide*, SC26-4767
- *COBOL Language Reference*, SC26-4769
- *COBOL Reference Summary*, SX26-3788

ISPF Publications

- *ISPF and ISPF/PDF Planning and Customizing*, SC34-4220
- *ISPF/PDF Guide*, SC34-4011
- *ISPF/PDF Services*, SC34-4012

PL/I for MVS & VM Publications

- *PL/I for MVS & VM Language Reference*, SC26-3114
- *PL/I for MVS & VM Migration Guide*, SC26-3118
- *PL/I for MVS & VM Programming Guide*, SC26-3113

Index

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

Printed in U.S.A.

SC09-7625-00