SAS/C C++ Development System User's Guide

Release 6.50

SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513

The correct bibliographic citation for this manual is as follows: SAS Institute Inc., *SAS/C C++ Development System User's Guide, Release 6.50*, Cary, NC: SAS Institute Inc., 1997. pp. 141.

SAS/C C++ Development System User's Guide, Release 6.50

ISBN 1–58025–134–X

SAS Institute Inc., SAS Campus Drive, Cary, North Carolina 27513.

1st printing, March 1998

Contents

PART 1

User’s Guide

CHAPTER 1

Introduction to the SAS/C C++ Development System

Introduction

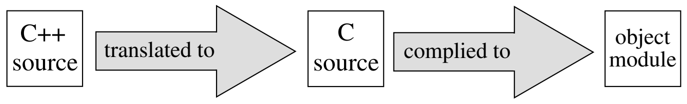
As an object-oriented programming language, C++ is an improved programming tool that makes program development, use, and maintenance easier and more efficient. The SAS/C C++ Development System enables you to develop and run C ++ programs on the mainframe under TSO, CMS, MVS batch, and OpenEdition.

This chapter provides an overview of the SAS/C C++ Development System and gives a brief description of the C++ language definition. After reading this chapter, you will be ready to begin using the SAS/C C++ Development System to develop your C++ programs.

Overview of the SAS/C C++ Development System

This release of the SAS/C C++ Development System implements the C++ language by means of a translator that translates C++ to C. The translated C code must be compiled with the SAS/C Compiler, resulting in an object module. The complete translation/compilation process for a C++ source program is shown in Figure 1.1 on page 3 .

Figure 1.1 Translation Process



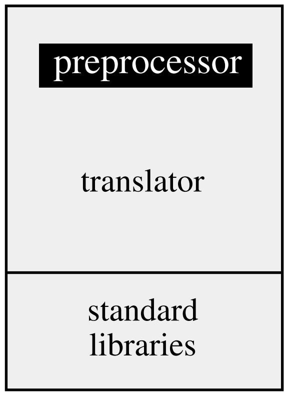
By default, when you invoke the translator both the translation and compilation are performed; that is, you do not have to call the SAS/C Compiler as a separate step. Also by default, the C source is not saved, but is instead a temporary file and is discarded after compilation. You can control the translation and compilation of your program by specifying options when you invoke the translator. These options are described in “Option Descriptions” on page 38.

Once you have translated all the source modules in your program, use the COOL utility to link the object modules into a load module. Using this load module, you can call your program as you would call any other executable file under your operating system.

SAS/C C++ Development System Components

As shown in Figure 1.2 on page 4 , the SAS/C C++ Development System consists of three parts: a preprocessor, the translator, and a set of standard libraries.

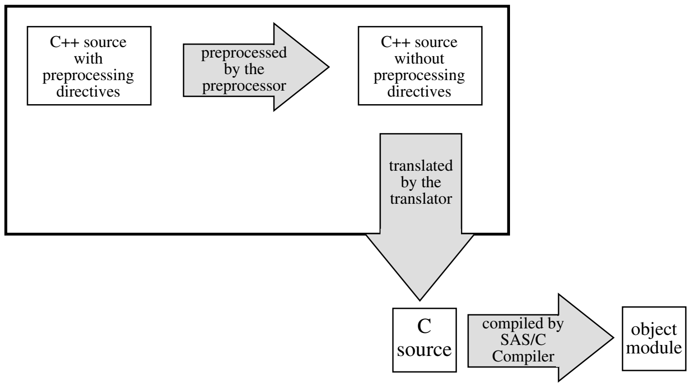
Figure 1.2 SAS/C C++ Development System Components



Preprocessor

As part of the translation process, your C++ code is preprocessed. Figure 1.3 on page 4 builds on Figure 1.1 on page 3 , showing the two separate phases the translator goes through before producing C source.

Figure 1.3 Expanded Translation Process


C++ source with preprocessing directives
preprocessed by the preprocessor
C++ source without preprocessing directives
translated by the translator
C source
compiled by SAS/C Compiler
object module

Note that the preprocessing and translating phases of the translator are conceptual and may not be implemented as separate physical steps.

The preprocessor is the part of the C++ translator that transforms its input text (C++ source code, possibly containing preprocessing directives and operators) to C++ text devoid of preprocessing directives and in which a variety of lexical substitutions or operations may have been performed.

After the preprocessor step is completed, the translator translates your C++ code into C code. (If you want to stop after the preprocessing step and skip the translation or compilation steps, specify the **pponly** option, which is described in "Option Descriptions" on page 38 .)

The preprocessor conforms to the ANSI Standard for C with the addition that it accepts C++ style comments and it correctly handles all C++ tokens. If you need more detailed information on a particular preprocessor feature, refer to the ANSI Standard for C or a C++ tutorial book.

Comments

The preprocessor supports both C++ style and C style comments. A C++ style comment begins with a double slash (//) and extends to the end of its line but not beyond. In contrast, a C style comment begins with a /* and ends with a * / and can take up multiple lines. For example, the following are some C++ style comments:

```
// This comment is too long to fit on one
// line, so it is continued on the second line.

int i; // declaration of an integer
```

Here are the equivalent C style comments:

```
/* This comment is too long to fit on one
   line, so it is continued on the second line.*/

int i; /* declaration of an integer */
```

In general, the comment style you use depends entirely on personal preference. If you want to embed a comment within a line of code, you must use a C style comment, as in the following example:

```
x=y; /* One assignment */ a=b; // and another.
```

However, imbedding comments like this does not produce readable code and is not recommended.

Translator

After your C++ code has been preprocessed, it is translated into C code. You can control the translation with a variety of options, which are discussed in "Option Descriptions" on page 38 .

Standard Libraries

This release of the SAS/C C++ Development System includes two standard libraries: the streams library and the complex library. Future releases of the product may include other libraries. The "C++ I/O" on page 53 and the "C++ Complex Library" on page 49 discuss the details of the functions and classes included in the streams and complex libraries.

C++ Language Definition

This section describes the main features of the SAS/C C++ Development System implementation. The discussion does not attempt to teach you C++ and assumes you have access to the *SAS/C Compiler and Library User's Guide, Fourth Edition*.

The C++ language accepted by the SAS/C C++ Development System is generally compatible with that specified by Bjarne Stroustrup in the *The C++ Programming Language, Second Edition*, with the exception of exception handling but with the addition of several ANSI C++ features. C++ code written for previous releases of the SAS/C C++ Development System generally will be accepted by Release 6.50. However, this release has increased compatibility with the ANSI/ISO C++ draft standard by implementing new features and by tightening language rules for certain obscure or unsafe constructs. Release 6.50 of the SAS/C C++ Development System includes these new ANSI C++ features:

- template support
- new cast operators
- Run-Time Type Identification or RTTI
- updated overload resolution rules
- support of condition declarations that allow expressions to be declared in the conditional part of **`if`** , **`switch`** , **`while`** , and **`for`** statements.
- Improved code generation, including:
  - expansion of inline functions without requiring the global optimizer
  - improved handling of temporaries. The translator will perform copy elimination on temporary values.
  - improved logic for creating the tables used to support virtual function calls. This can result in large space savings, especially in libraries.

Because C++ is in general a superset of C, the C++ language includes many SAS/C features. You can find a detailed definition of the SAS/C implementation in "Language Definition" in the *SAS/C Compiler and Library User's Guide*. This section describes only those features that are different from SAS/C behavior or that are specific to the C++ environment.

If you have a C++ program that is compliant with AT&T C++ version 3.0, your code generally should work with the SAS/C C++ Development System.

If you have a C++ program that is compliant with AT&T C++ version 2.1, with the exception of some features that are now considered anachronistic, your code generally should work with the SAS/C C++ Development System. The main differences between AT&T C++ 2.1 and the SAS/C C++ Development System are the following:

- SAS/C C++ does not support overloading both the prefix and postfix ++ or -- operators with a single function.
- SAS/C C++ does not support the transitional model of nested classes. That is, a class that is nested is always treated as nested, even if there is not a file-scope class of the same name.

For a complete list of which anachronisms are supported by the SAS/C C++ Development System, see "Anachronisms" on page 11 .

Incompatibility with Previous Releases

Object code generated by Release 6.50 of the C++ translator is not compatible with object code generated by previous releases of the C++ translator and is not compatible with the C++ library for previous releases.

Environmental Elements

This section describes four important environmental elements:

- special characters
- storage class limits
- numerical limits
- source file sequence numbers.

Special characters

C++ uses a number of special characters. Many IBM mainframe terminals and printers do not supply all of these characters. The SAS/C C++ Development System provides two solutions to this problem:

- a special character translation table
- digraphs (described in Table 1.5 on page 9).

The special character translation table enables each site to customize the representation of special characters. That is, sites can decide which hexadecimal bit pattern or patterns represent that character and so can choose a representation that is available on their terminals and printers.

The special characters that can be customized are braces, square brackets, circumflex, tilde, backslash, vertical bar, pound sign, and exclamation point. You should determine if your site has customized values for these characters and find out what the values are. Otherwise, the default representations listed in Table 1.1 on page 5 are in effect. Consult your SAS Software Representative for details about customized values. Table 1.1 on page 5 shows the two possible default representations for each character. These primary and alternate representations in columns two and three are EBCDIC equivalents of the characters in hexadecimal notation.

Remember that the alternate representations for characters apply only to C++ program source code and not to general file contents read by C++ programs.

Table 1.1 Default Representations for Special Characters

	Source File Representation	
Character	Primary	Alternate
left brace {	0xc0 {	0x8b {
right brace }	0xd0 }	0x9b }
left bracket [	0xad [	0xad [
right bracket]	0xbd]	0xbd]
circumflex ^ (exclusive or)	0x5f ¬	0x71 ^
tilde ~	0xa1 ~	0xa1 ~
backslash \	0xe0 \	0xbe ≠
vertical bar | or ¦ (inclusive or)	0x4f |	0x6a ¦

	Source File Representation	
Character	Primary	Alternate
pound sign #	0x7b #	0x7b #
exclamation point !	0x5a !	0x5a !

Storage class limits

The SAS/C Compiler imposes several limits on the sizes of various objects and these may affect your C++ program after it is translated into C and compiled.

The total size of all objects declared in one translation with the same storage class is limited according to the particular storage class, as follows:

extern	16,777,215 (16M-1) bytes
static	8,388,607 (8M-1) bytes
auto	8,388,607 (8M-1) bytes
formal	65,535 (64K-1) bytes.

Individual objects can be up to 8 megabytes in size. The translator imposes no limit on array sizes.

The following types of programs generate very large CSECTS:

- programs compiled with **`norent`** and with large amounts of **`static`** or defined external data or both
- programs compiled with **`rentext`** and with large amounts of **`static`** data.

You should consider alternatives to using large amounts of **`static`** data. One alternative is to use the **`new`** operator for dynamic storage allocation. Storage allocated with the new operator is limited only by available memory.

Numerical limits

The numerical limits are what one would expect for a 32-bit, twos complement machine such as the IBM 370. Table 1.2 on page 6 shows the size ranges for the integral types.

Table 1.2 Integral Type Sizes

Type	Length in Bytes	Range
`char`	1	0 to 255 (EBCDIC character set)
`signed char`	1	-128 to 127
`short`	2	-32,768 to 32,767
`unsigned short`	2	0 to 65,535
`int`	4	-2,147,483,648 to 2,147,483,647
`unsigned int`	4	0 to 4,294,967,295
`long`	4	-2,147,483,648 to 2,147,483,647
`unsigned long`	4	0 to 4,294,967,295

Table 1.3 on page 6 shows the size ranges for float and double types.

Table 1.3 Float and Double Type Sizes

Type	Length in Bytes	Range
`float`	4	+/- 5.4E-79 to +/- 7.2E75
`double`	8	+/- 5.4E-79 to +/- 7.2E75
`long double`	8	+/- 5.4E-79 to +/- 7.2E75

Additional details on the implementation of the various data types can be found in "Compiler Processing and Code Generation Conventions," in the *SAS/C Compiler and Library User's Guide*.

Source file sequence number handling

The translator examines the first record in the source file and each #include file to determine if the file contains sequence numbers. Therefore, you can safely mix files with and without sequence numbers and use the translator on sequenced or nonsequenced files without worrying about specifying a sequence number parameter.

For a file with varying-length records, if the first four characters in the first record are alphanumeric and the following four characters are numeric, then the file is assumed to have sequence numbers.

For a file with fixed-length records, if the last four characters in the first record are all numeric and the preceding four characters are alphanumeric, then the file is assumed to have sequence numbers.

If a file is assumed to have sequence numbers, then the characters in each record at the sequence number position are ignored. This algorithm detects sequence numbers or their absence correctly for almost all files, regardless of record type or record length. Occasionally the algorithm may cause problems, as in the following examples:

- For a file in which only some records, not including the first record, contain sequence numbers, the validity of the sequence number is questionable. The entire record is treated as C code, so errors are certainly generated.
- A file of fixed-length records in which the last eight characters of the first record resemble a sequence

number but are instead, for example, a long numeric constant also causes a problem. A dummy first record or a comment after the digits fixes this problem.

Language Elements

Certain language elements, such as constants and predefined constants, deserve special explanation, as the translator treats them in accordance with the language described in *The C++ Programming Language*.

Constants

This section describes how the translator treats character constants and string literals.

Character constants

The translator produces a unique **char** value for certain alphabetic escape sequences that represent nongraphic characters. This **char** value corresponds to the hex values shown in column 2 of Table 1.4 on page 7 .

Table 1.4 Escape Sequence Values

Sequence	Hex Value	Meaning
\a	0x2f	alert
\b	0x16	backspace
\f	0x0c	form feed
\n	0x15	newline
\r	0x0d	carriage return
\t	0x05	horizontal tab
\v	0x0b	vertical tab

String literals

By default, identically written string constants refer to the same storage location: only one copy of the string is generated by the translator. The **NOSTringdup** compiler option can be used to force a separate copy to be generated for each use of a string literal. However, modifying string constants is not recommended and renders a program nonreentrant.

Note: Strings used to initialize **char** arrays (not **char***) are not actually generated because they are shorthand for a comma-separated list of single-character constants. △

Predefined constants

The translator supports several predefined constants:

- **__cplusplus**
- **c_plusplus**
- **__DATE__**
- **__FILE__**
- **__LINE__**
- **__TIME__**

These macros are useful for generating diagnostic messages and inline program documentation. The following list explains the meaning of each macro:

__cplusplus
expands to the decimal constant 1.

c_plusplus
expands to the decimal constant 1.

__DATE__
expands to the current date, in the form *Mmm dd yyyy* (for example, Jan 01 1990). Double quotes are a part of the expansion; no double quotes should surround **__DATE__** in your source code.

__FILE__
expands to a string literal that specifies the current filename. Double quotes are a part of the expansion; no double quotes should surround **__FILE__** in your source code.

For the primary source file under MVS batch, **__FILE__** expands to the data set name of the source file, if it is a disk data set, or the DDname allocated to the source file. For the primary source file under CMS, **__FILE__** expands to "filename filetype", where filename is the CMS filename and filetype is the CMS filetype.

For a **#include** or header file, under both MVS and CMS, **__FILE__** expands to the name that appears in the **#include** statement, including the angle brackets or double quotes as part of the string. Thus, for the following, **__FILE__** expands to "\" **myfile.h** \"":

```
#include "myfile.h"
```

For the following, **__FILE__** expands to " **<myfile h e>** ":

```
#include <myfile h e>
```

__LINE__
expands to an integer constant that is the relative number of the current source line within the file (primary source file or #include file) that contains it.

__TIME__
expands to the current time, in the form **hh:mm:ss** (for example, **10:15:30**). Double quotes are a part of the expansion; no double quotes should surround **__TIME__** in your source code.

None of the above predefined macros can be undefined with the **#undef** directive.

The translator also provides the following predefined macro names. Automatic predefinition of these names can be collectively suppressed by using the **undef** translator option. (Refer to “Option Descriptions” on page 38 for more information on **undef** .) These macro names also can be undefined by the **#undef** preprocessor directive.

The following code shows their usage:

```
#define OSVS 1    // if translating under TSO
                  // or MVS batch
#define CMS 1     // if translating under CMS
#define I370 1    // indicates the SAS/C
```

```
                 // Compiler or the translator
#define DEBUG 1  // if the DEBug option is
                 // used
#define NDEBUG 1 // if the DEbug option is not used
```

A few of the predefined macros can only be undefined by the **#undef** preprocessor directive. They are not affected by the **undef** translator option. These macros are:

```
#define __COMPILER__ "SAS/C C++ 6.50B" // indicates
      // the current release
      // as a string
#define __I370__ 1   // indicates the SAS/C
      // Compiler or the translator
#define __SASC__ 650 // indicates the current
      // version as a number,
      // for example, 650
```

Note: Because the translator is not a C compiler, the **__STDC__** macro is not defined. △

Language Extensions

This section describes SAS/C extensions to the language described in *The C++ Programming Language.*

Note: Use of these extensions is likely to render a program nonportable. △

For information on SAS/C extensions to the C language, such as the **__asm** keyword, the **__alignmem** and **__noalignmem** keywords, and keywords used in declarations of functions that are neither C++ nor C, see the *SAS/C Compiler and Library User's Guide, Fourth Edition*. Also refer to the *SAS/C Compiler and Library User's Guide* for a discussion of the implementation-defined behavior of the SAS/C Compiler.

Preprocessor extensions

Two **#pragma** directives are handled by the SAS/C C++ Development System directly:

```
#pragma linkage
#pragma map
```

These **#pragma** directives are described in the *SAS/C Compiler and Library User's Guide*. In C++ programs, these directives can be applied only to functions and variables that have **extern** "C" linkage (that is, they are declared in an extern "C" block or have extern "C" in their declaration).

The **__ibmos** SAS/C extension keyword is a simpler and more direct replacement for **#pragma linkage** . The **__ibmos** keyword is described in the *SAS/C Compiler and Library User's Guide*. AR370 is a simpler and more powerful replacement for **#pragma map** . The AR370 utility is described in the *SAS/C Compiler and Library User's Guide*.

All other **#pragma** directives are passed on directly to the output C file and are otherwise ignored by C++.

SAS/C extension keywords

You can use the following SAS/C extension keywords in your C++ programs:

```
__asm        __local      __weak
__cobol      __pascal
__foreign    __pli
__fortran    __ref
__ibmos      __remote
```

Overloading on these SAS/C extension keywords is supported. The following example shows overloading **error_trap** to take both local and remote function pointers:

```
int error_trap(_\x12_local void(*f)());
int error_trap(_\x12_remote void(*f)());
```

Functions defined using one or more of the keywords **__ibmos** , **__asm** , or **__ref** must be written in assembler. Therefore, the translator assumes "C" linkage for these functions, even if extern "C" is not explicitly used. Similarly, **__pli** , **__cobol** , **__fortran**, **__pascal**, and **__foreign** functions have linkage appropriate for the language and therefore do not have C++ linkage. The main effect of this behavior is that overloading the following functions is not allowed:

```
__asm int myfunc(int);
__pli int myfunc(int*);
```

These functions cannot be overloaded because only one linkage version of a function that is not C++ is permitted.

For more information on SASC extension keywords, see the *SAS/C Compiler and Library User's Guide*.

Alternate forms for operators and tokens

C++ is traditionally implemented using the ASCII character set. The translator uses EBCDIC as its native character set because EBCDIC is the preferred character set under TSO, CMS, and MVS batch. Because some characters used by the C++ language are not normal EBCDIC characters (that is, they do not appear on many terminal keyboards), alternate representations are available. Also, for some characters, there is more than one similar EBCDIC character. The translator accepts either character.

Table 1.5 on page 9 lists alternate representations that the translator accepts (this set of digraphs is identical to the digraph set accepted by the SAS/C Compiler). The digraph option(s) chosen determines which alternate forms are used:

digraph option 1
: turns on the new ISO digraph support.

digraph option 2
: turns on SAS/C bracket digraph support, '(|' or '|)'.

digraph option 3
: turns on all SAS/C digraphs but does not activate the new ISO digraphs unless option 1 is also activated.

See “Option Descriptions” on page 38 for more information on digraph options.

Table 1.5 Digraph Sequences for Special Characters

C++ Character	EBCDIC Value(s) (hex)	Alternate Forms (for use with digraph options 2, 3)	Alternate Forms (for use with digraph option 1)
[(left bracket)	0xad	(|	<:
] (right bracket)	0xbd	|)	:>
{ (left brace)	0x8b, 0xc0	\\(or (<	<%
} (right brace)	0x9b, 0xd0	\\) or >)	%>
| (inclusive or)	0x4f, 0x6a	\\!	
~ (tilde)	0xa1	\\¬	
#(pound sign)	0x7b		%:
##(double pound sign)	0x7b 0x7b		%:%:
\\ (backslash)	0xe0, 0xbe	(see below)	

For all symbols except the backslash, substitute sequences are not replaced in string constants or character constants. For example, the string constant "<:" contains two characters, not a single left bracket. Contrast this behavior with the ANSI trigraphs, which are replaced in string and character constants.

The backslash is a special case because it has meaning within string and character constants as well as within C++ statements. You can also customize the translator to accept an alternate single character for the backslash, as well as for other characters in Table 1.5 on page 9 . The default alternate representations are listed in Table 1.1 on page 5 . See your SAS Software Representative for more information.

Embedded $ in identifiers

The dollar sign ($) can be used as an embedded character in identifiers. If the dollar sign is used in identifiers, the **`dollars`** translator option must be specified. Use of the dollar sign is not portable because the dollar sign is not part of the portable C++ character set. The dollar sign cannot be used as the first character in an identifier; such usage is reserved for the library.

Floating-point constants in hexadecimal

An extended format for floating-point constants enables them to be specified in hexadecimal to indicate the exact bit pattern to be placed in memory. A hexadecimal **`double`** constant consists of the sequence **`0.x`** , followed by 1 to 14 hexadecimal digits. If there are fewer than 14 digits, the number is extended to 14 digits on the right with 0s. A hexadecimal **`double`** constant defines the exact bit pattern to be used for the constant. For example, **`0.x411`** has the same value as **`1.0`** . Use of this feature is nonportable.

Call-by-reference operator (@)

The `@` operator is a language extension provided primarily to aid communication between C++ and other programs.

In C++ (as in C), the normal argument-passing convention is to use call-by-value; that is, the value of an argument is passed. The normal IBM 370 (neither C nor C++) argument-passing conventions differ from this in two ways. First, arguments are passed by reference; each item in the parameter list is an argument address, not an argument value. Second, the last argument address in the list is usually flagged by setting the high-order bit.

One approach to the call-by-reference problem is to precede each function argument by the `&` operator, thereby passing the argument address rather than its value. For example, you can write **`asmcode(&x)`** rather than **`asmcode(x)`** . This approach is not generally applicable because it is frequently necessary to pass constants or computed expressions, which are not valid operands of the address-of operator. The translator provides an option to solve this problem.

When the translator option **`at`** is specified, the at sign (`@`) is treated as an operator. The `@` operator can be used only on an argument to a function call. The result of using it in any other context is undefined. The `@` operator has the same syntax as the C ampersand (`&`) operator. In situations where the C `&` can be used, `@` has the same meaning as `&` . In addition, `@` can be used on values that are not lvalues such as constants and expressions. In these cases, the value of `@` expr is the address of a temporary storage area to which the value of expr is copied. One special case for the `@` operator is when its argument is an array name or a string literal. In this case, **`@array`** is different from **`&array`** . The latter still addresses the array, while **`@array`** addresses a pointer addressing the array. Use of `@` is, of course, nonportable. Its use should be restricted to programs that call routines, that are not C++, .using call-by-reference.

When declaring a call by reference instead of using the `@` notation, you may want to use the **`__asm`** or **`__ref`** keyword described in the *SAS/C Compiler and Library User's Guide*.

Nesting of #define

If the **`redef`** translator option is specified, multiple **`#define`** statements for the same symbol can appear in a source file. When a new **`#define`** statement is encountered for a symbol, the old definition is stacked but is restored if an #undef statement for the symbol occurs. For example, if the line

```
#define XYZ 12
```

is followed later by

```
#define XYZ 43
```

the new definition takes effect, but the old one is not forgotten. Then, when the translator encounters the following, the former definition (12) is restored:

```
#undef XYZ
```

To completely undefine XYZ, an additional **`#undef`** is required. Each **`#define`** must be matched by a corresponding **`#undef`** before the symbol is truly forgotten. Identical **`#define`** statements for a symbol (those permitted when **`redef`** is not specified) do not stack.

Zero-length arrays

An array of length 0 can be declared as a member of a structure or class. No space is allocated for the array, but the following member is aligned on the boundary required for the array type. Zero-length arrays are useful for aligning members to particular boundaries (to match the format of external data for example) and for allocating varying-length arrays following a structure. In the following structure definition, no space is allocated for member **`d`** , but the member b is aligned on a doubleword boundary:

```
struct ABC
{
   int a;
   double d[0];
   int b;
};
```

Zero-length arrays are not permitted in any other context.

__inline and __actual storage class modifiers

`__inline` is a storage class modifier. It can be used in the same places as a storage class specifier and can be declared in addition to a storage class specifier. If a function is declared as **`__inline`** and the module contains at least one definition of the function, the translator sees this as a recommendation that the function be inlined. If a function is declared as **`__inline`** and has external linkage, a real copy of the function is created so that other external functions can call it.

With the 6.50 release, if you use inline functions and have **`DEBUG`** turned off, the translator performs inlining of inline functions whether the **`optimize`** option is on or off. If **`DEBUG`** is turned on, the translator disables inlining. The **`optimize`** option is on by default.

`__actual` is also a storage class modifier. It can be specified with or without the **`__inline`** qualifier, but it implies **`__inline`** . **`__actual`** specifies that the translator should produce an actual (callable) copy of the function if the function has external linkage. If the function has internal linkage, the translator creates an actual function unless it does not need one.

For additional information, see the discussion of **`__inline`** and **`__actual`** in the *SAS/C Compiler and Library User's Guide*.

Note: The difference between the **`__inline`** modifier and the **`inline`** C++ keyword is that the inline keyword causes inline functions to behave as if they were declared static while **`__inline`** does not. In some cases, current ANSI C++ rules may treat the inline function as if it has external linkage. △

Implementation-Defined Behavior

Implementation-defined behaviors are translator actions that are not explicitly defined by the language standard. For example, in *The C++ Programming Language*, Stroustrup leaves the range of values that can be accommodated by a double to the discretion of individual implementations. Each implementation is simply required to document the chosen behavior. Allowing implementation-defined behavior enables each vendor to implement the C++ language as efficiently as possible with in the particular operating system and environment. This section describes the implementation-defined behaviors of the translator.

Much of the implementation-defined behavior of the translator corresponds to the implementation-defined behavior of the SAS/C Compiler, while some behaviors are specific to C++. The next two sections describe the implementation-defined behavior of the translator in detail.

Behaviors related to the SAS/C Compiler

The following list enumerates those behaviors common to both the translator and compiler or behaviors that are similar but have small differences. The following list gives a brief description of the behavior and a reference to the *SAS/C Compiler and Library User's Guide*, where necessary.

- Characteristics of fundamental types for the translator are the same as those for the SAS/C Compiler. These characteristics, defined in the **`limits.h`** and **`float.h`** C header files, are described in the *SAS/C Compiler and Library User's Guide*.
  - Alignment requirements of fundamental types for the translator are the same as those for the SAS/C Compiler. These alignment requirements are also described in the *SAS/C Compiler and Library User's Guide*.
  - The translator does not support multibyte character constants and **`wchar_t`** initializers.
  - The translator treats duplicate string constants in the same way as the SAS/C Compiler. This topic is discussed in the *SAS/C Compiler and Library User's Guide*. (By default, only one copy of a string literal is kept at one time.)
  - The translator treats arithmetic overflow and division by zero in the same way as the SAS/C Compiler. This topic is discussed in the *SAS/C Compiler and Library User's Guide*. (By default, integer overflow is ignored and both floating-point overflow and division by zero cause abnormal program termination unless an arithmetic signal handler is defined.)
  - The type of **`size_t`** is **`unsigned int`** , as it is for the SAS/C Compiler.
  - The type of **`ptrdiff_t`** is **`signed long`** .

- □ The translator maps pointers to and from integers the same way as the SAS/C Compiler. This topic is discussed in *Implementation-defined Behavior* in the *SAS/C Compiler and Library User's Guide*.
- □ As with the SAS/C Compiler, the remainder from integer division (using the binary % operator) has the same sign as the dividend, except that a 0 remainder is always positive.
- □ Right shifting a negative integral value works the same as it does in SAS/C software; that is, the sign bit is used to fill the vacated positions on the left. The result retains the sign of the first operand.
- □ The alignment and sign of bitfields, both plain and noninteger, works the same as it does for the SAS/C Compiler. These topics are discussed in the *SAS/C Compiler and Library User's Guide*.

  To summarize, the **`bitfield`** translator option causes the translator to accept any integral type in the declaration of a bitfield and enables you to control the allocation of bitfields. By default, bitfields are aligned on word boundaries. Plain **`int`** bitfields are treated as **`unsigned int`** bitfields, and the order of allocation of bitfields with an **`int`** is left to right.
- □ The details of how the translator and compiler search for **`#include`** header files are the same as for the SAS/C Compiler. This topic is discussed in the *SAS/C Compiler and Library User's Guide*. You should also refer to Chapter 2, "Using the SAS/C C++ Development System under TSO, CMS, MVS Batch, and OpenEdition," on page 13 .
- □ The meaning of the **`#pragma`** preprocessing directive is discussed earlier in this chapter, in "C++ Language Definition" on page 4 .
- □ The type of **`main`** is the same as it is for the SAS/C Compiler and complies with the ANSI definition.

  To summarize, main can be defined in one of two ways. It can either take no arguments and be defined as

  ```
  int main(void){ /* ... */ }
  ```

  Or, it can have two parameters and be defined as

  ```
  int main(int argc, char *argv[])
          { /* ... */ }
  ```

  For more information on the constraints for argc and argv, see Section 2.1.2.2 of the ANSI C Standard.

  You should also refer to the *SAS/C Compiler and Library User's Guide* for information about the environment variables extension provided by the SAS/C Compiler.

C++-specific behaviors

Some implementation-defined behavior is specific to the C++ language. The following list enumerates these behaviors.

- □ There are only two accepted linkage strings, " **`C`**" and " **`C++`**" . " **`C`** " linkage for functions means that type information is not encoded in the function's identifier. " **`C`**" linkage does not affect the linkage for nonfunctions. Remember that the linkage for **`main`** is always " **`C`** ".
- □ **`operator new`** is responsible for memory allocation; it does not leave the allocation up to the constructor.
- □ The effect of modifying a **`const`** object through a non-**`const`** pointer is unpredictable and could cause an ABEND.
- □ An **`asm`** declaration (for example, **`asm(str_lit)`**) is not allowed by the translator. Do not confuse the use of an **`asm`** declaration with the SAS/C **`__asm`** keyword.
- □ Base classes are allocated in the order in which they are specified in the derived class.
- □ Non **`static`** data members are allocated in the order in which they are declared.
- □ There is no nesting limit for **`#include`** statements or for conditional compilation.

Initialization and termination

This section describes the order of initialization and termination of file-scope objects defined in C++. Initialization of an object consists of executing its initializer. This includes executing the object's constructor if it has one. Termination of an object consists of executing the object's destructor, if it has one. In general, objects are terminated in the reverse of the order that they are initialized, but this is not necessarily the case for objects in dynamically loaded modules.

When a program containing C++ code is started, file-scope objects defined in C++ translation units in the main load module are initialized in the reverse order of the translation unit's inclusion into the load module by COOL. (For more information on this topic, see "INCLUDE Statement" on page 29 .) Within a translation unit, objects are initialized in the order that they are defined in the translation unit.

When the main program ends, either by calling **`exit`** or by returning from **`main`** , file-scope objects defined in C++ translation units in the main load module are terminated in the reverse order of how they were initialized.

Objects defined in C++ translation units in dynamically loaded modules are initialized when the module is loaded and terminated when the module is unloaded. Within a dynamically-loaded module, the order of initialization and termination is the same as for the main load module.

Anachronisms

The following list enumerates several features that a C++ implementation may provide to support old coding styles and features. The following list enumerates these compatibility issues and indicates which are supported by the SAS/C C++ translator.

- □ The **`overload`** keyword is supported but must be enabled by the **`overload`** translator option. **`overload`** is not treated as a reserved word unless you turn on the **`overload`** option.
- □ The **`redef`** option allows you to nest **`#define`** statements. For more details, see "Nesting of #define" on page 9 .

- Kernighan and Ritchie (K&R) C style function definitions are allowed.
- Cast of a bound pointer is not supported.
- Assignment to **`this`** is not supported.
- The number of elements in an array can be specified when deleting an array; however, this number is ignored.
- In some implementations, a single function **`operator ++()`** can be used to overload both prefix and postfix ++ and a single function **`operator --()`** can be used to overload both prefix and postfix **`--`** . The SAS/C C++ Development System follows the more modern practice of using different function signatures for overloading the prefix and postfix forms of these operators.
- Nested class tags are hidden by the surrounding class or structure. That is, if you declare a class within another class, even if no other class of that name is declared in your program, you cannot use the nested class tag as if it was declared outside its enclosing class.

CHAPTER

2

Using the SAS/C C++ Development System under TSO, CMS, MVS Batch, and OpenEdition

Introduction

This chapter explains how to use the translator, the SAS/C Compiler, the COOL object code preprocessor, and a number of operating system utilities to create and run C++ programs. Each section contains operating environment specifics that you need to know when you are developing and running your programs under TSO, CMS, MVS Batch, and OpenEdition.

However, you can develop a C++ program in one of these environments and use it in another. In this case, you will need to read all of the appropriate sections. For example, it is very common to develop a program using MVS cataloged procedures, but run the finished program only as a TSO command. In this case, you should read about running the translator, compiler, and linker in “Creating C++ Programs under MVS Batch” on page 19 and about running the programs in “Creating C++ Programs under TSO” on page 13.

Creating C++ Programs under TSO

Before running the translator, compiler, COOL, or any C++ program, ensure that the transient library is allocated to the CTRANS DDname or that it is installed in the system link list. Consult your SAS Software Representative to determine if this has been done for you.

Translating and Compiling Your Program under TSO

The LCXX CLIST invokes the translator and compiler. Optionally, you can also invoke the OMD370 object module disassembler utility. The LCXX CLIST has the following format:

LCXX *dsname<options>*

where *dsname* is the name of the data set containing the C++ program to be translated and compiled. Follow standard TSO data set naming conventions. If the data set belongs to another user, specify the full data set name enclosed in apostrophes. If the program is a member of a partitioned data set (PDS), specify the member name in parentheses following the data set name. Here are two examples of invoking LCXX to translate and compile a program in a data set belonging to another user:

```
LCXX 'FRIEND.PROJ.CXX'
LCXX 'LEADER.APPL.CXX(SUBRTN)'
```

If the data set belongs to you and you do not enclose the data set name in apostrophes, LCXX assumes that the final qualifier of the data set name is CXX. If you do not specify CXX, the CLIST adds it for you. Here are two examples of invoking LCXX to translate and compile a program in a data set belonging to you:

```
LCXX PROJ.CXX
LCXX APPL(SUBRTN)
```

In the second example, the CLIST assumes that the program is in *userid*.APPL.CXX(SUBRTN). If the data set belongs to you but is not named according to the CLIST assumptions, you must enclose the data set name in apostrophes, as if the data set belonged to someone else. *dsname* must be the first item on the command line. *options* are any translator, compiler, and, if the OMD option is used to invoke the OMD370 utility, OMD370 options. Separate the options with one or more blanks, commas, or tabs. See “Option Summary” on page 35 for the options that are available for use with the translator. The LCXX CLIST executes the compiler if no errors occur during translation. LCXX allocates a temporary data set to contain the translator output.*

Saving the output data set under TSO

The LCXX CLIST supports three options that enable you to save intermediate code produced by the translator: **pponly**, **tronly**, and **savec**. These options require you to specify the name of an output data set, using standard TSO conventions. Unlike the input data set name, LCXX does not assume a final qualifier for the name of the output data set. Also, if the data set name is fully qualified, use three apostrophes on each end. Here is a brief description of each of the **pponly**, **tronly**, and **savec** options:

pponly	performs only the preprocessing step. The program is not translated or compiled. The output data set contains C++ code.
tronly	performs the preprocessing and translation steps. The program is not compiled. The output data set contains C code.
savec	saves the C code resulting from translation. The program is preprocessed, translated, and compiled. The output data set contains C code.

For detailed information on these three options, see “Option Descriptions” on page 38.

Note: Because both **tronly** and **savec** save the intermediate C code data set, you should use only one of them. For example, if you use **tronly**, you do not need to also specify **savec**. △

Here are some examples of specifying an output data set. In this first example, the **pponly** option causes only the preprocessing phase to be executed and the C++ code to be saved in *userid*.PROJ.PP(MAIN). The C++ code is not translated or compiled.

```
LCXX PROJ(MAIN) PPONLY(PROJ.PP(MAIN))
```

In this next example, the **tronly** option causes the translator to be invoked and the resulting C code to be saved in *userid*.PROJ.C(MAIN); however, the C code is not compiled.

```
LCXX 'userid.PROJ.PP(MAIN)' TRONLY(PROJ.C(MAIN))
```

In this final example, the savec option causes the translator and compiler to be invoked. The intermediate C code is saved in LEADER.PROJ.C(MAIN).

```
LCXX 'userid.PROJ.PP(MAIN)'
   SAVEC('''LEADER.PROJ.C(MAIN)''')
```

Locating header files under TSO

When a header filename is surrounded by angle brackets in a **#include** statement, the translator searches for the header file in standard C and C++ header files supplied by SAS Institute and in any library specified via the LCXX CLIST **LIB** option. The syntax for this form of the **#include** statement is

```
#include <member.ext>
```

The *.ext* part can be omitted. Here is an example of including a standard header file:

```
#include <new.h>
```

For this statement, the translator includes the file named NEW from the standard C++ header files.

* The name of this data set is in the form *userid*.SASCTEMP.$*nnnnnn*.C where *nnnnnn* is a sequence of decimal digits. This temporary data set is deleted after the compiler has completed. If by chance this file does not get deleted by default, you can delete it yourself.

When you use the LCXX CLIST, the data sets that contain the standard C++ and C library header files are automatically available. You can add your own data sets to this concatenation via the LCXX CLIST **LIB** option.

To include a header file from your personal files (as opposed to header files from system files), surround the filename with double quotes, as shown here:

```
#include "member.ext"
```

The translator assumes that the header file is ***member*** in a PDS allocated to the ***ext*** DDname. You must allocate this DDname to the ***member*** file data set before invoking the LCXX CLIST. The ***.ext*** part can be omitted, in which case it defaults to **.h**. Here is an example:

```
#include "project.h"
```

For this statement, the translator includes member PROJECT from the file associated with the DDname H. If the file is not found, the translator also searches the system header files.

Linking Your Program under TSO

All C++ programs must be preprocessed by the COOL object code preprocessor because COOL automatically creates object code used during the initialization and termination of **static** objects. In addition, COOL supports linking object files that contain mixed case external names longer than eight characters, such as those created by the compiler for source code generated by the translator. Such external names cannot be handled by the linkage editor unless the object code has been preprocessed by COOL.

COOL CLIST

The COOL CLIST invokes COOL to preprocess your object code and then calls the linkage editor to create a load module. The format of the COOL CLIST is

COOL *dsname* CXX <*options*>

where *dsname* is the name of the primary input data set and it is required. The data set must contain either object code or control statements or both. Follow standard TSO data set naming conventions when specifying this name. If the data set belongs to another user, specify the full data set name enclosed in apostrophes. If the program is a member of a PDS, specify the member name in parentheses following the data set name. The following two examples invoke COOL to link a program in a data set belonging to another user:

```
COOL 'FRIEND.PROJ.OBJ' CXX
COOL 'LEADER.APPL.OBJ(SUBRTN)' CXX
```

If the data set belongs to you and you do not enclose the data set name in apostrophes, COOL assumes that the final qualifier of the data set name is OBJ. If you do not specify OBJ, the CLIST adds it for you. The following two examples invoke COOL to link a program in a data set belonging to you:

```
COOL PROJ.OBJ CXX
COOL APPL(SUBRTN) CXX
```

In the second example, the CLIST assumes that the program is in *userid*.APPL.OBJ(SUBRTN). If the data set belongs to you but is not named according to the CLIST assumptions, you must enclose the data set name in apostrophes, as if the data set belonged to someone else. *dsname* must be the first item on the command line.

CXX is the only required option. This option makes the standard C++ object library SASC.LIBCXX.A, as well as the standard C object libraries, available to COOL. You can add your own data sets to this concatenation via the COOL **LIB** option. *options* are any COOL or linkage editor options. Separate the options with one or more blanks, commas, or tabs. Refer to "COOL Options" on page 30 for more information.

COOL also accepts input from AR370 archives. For information on this type of file, refer to the *SAS/C Compiler and Library User's Guide*.

COOL input files can contain control statements instead of, or in addition to, object code. Refer to "COOL Control Statements" on page 29 for more information.

More detailed information about the COOL utility and the COOL CLIST is available in the *SAS/C Compiler and Library User's Guide*.

Running Your Program under TSO

C++ programs can be called under TSO via the TSO CALL command. Depending on how the SAS/C C++ Development System has been installed at your site, you may have a higher level of support available. The optional methods are

- □ calling via the C command
- □ calling as a standard TSO command.

Here are some examples. Suppose that the load module for your program is member CXXPROG in the data set *userid*.APPL.LOAD. Suppose that you want to redirect **stdin** to the data set allocated to the DDname INPUT and pass the program option **-z**. Finally, suppose that you want to override the default initial stack allocation using the run-time option **=48k**. You can call the program using the CALL command as shown here:

```
CALL APPL(CXXPROG) '<INPUT -Z =48K'
```

The CALL command automatically translates program arguments to uppercase. If your program requires lowercase arguments, you can use the **ASIS** option of the CALL command to suppress uppercasing of arguments. Here is an example using the **ASIS** option:

```
CALL APPL(CXXPROG) '<INPUT -z =48k' ASIS
```

Some older versions of TSO do not support the ASIS option.

If you want to run your program with the SAS/C Debugger, use the **=D** option when you call your program, as in the following example:

```
CALL APPL(CXXPROG) '=D'
```

For more information on debugging C++ programs, refer to Chapter 5, "Debugging C++ Programs Using the SAS/C Debugger," on page 97.

Note: Programs called via the CALL command cannot access their name via the pointer in **argv[0]**. △

If the C command has been installed at your site, allocate your program library *userid*.APPL.LOAD to the DDname CPLIB and call the program using this command line:

```
C CXXPROG <INPUT -Z =48K
```

Also, if your site supports calling C programs as standard TSO commands, you can call your program using this command line:

```
CXXPROG <INPUT -Z =48K
```

This method also requires that your program library *userid*.APPL.LOAD be allocated to the DDname CPLIB.

Support for program invocation other than via the TSO CALL command is optional. Consult your SAS Software Representative to determine if this support is available at your site.

Creating C++ Programs under CMS

Before running the translator, compiler, COOL, or any C++ program, ensure that the transient library is available on an accessed minidisk or that it is installed in a segment available to your virtual machine. Consult your SAS Software Representative to determine if this has been done for you.

Translating and Compiling Your Program under CMS

The LCXX EXEC invokes the translator and compiler. Optionally, you can also invoke the OMD370 object module disassembler utility. The LCXX EXEC has the following format:

LCXX *fileid* <(*options*<)>>

where *fileid* is the fileid of the file to be translated and compiled. The fileid can name a file on a CMS minidisk, a file in the XEDIT ring, or a file in a Shared File System (SFS) directory. The format of the fileid is described in "Specifying a fileid" on page 16. The *options* are any translator options, compiler options, or, if the OMD option is used to invoke the OMD370 utility, OMD370 options. See "Option Summary" on page 35 for the options that are available for use with the translator.

By default, the translator writes its output (C code) to a temporary file and runs the compiler if no errors were found. The name of the output file has the same filename as the input file and a filetype of TROUT. If the input file is on an accessed minidisk or directory accessed as a minidisk, the temporary output file is written on that minidisk. If the minidisk is not write-accessed, the output file is written to the minidisk accessed as filemode A. If the input file is in an SFS directory that is not accessed as a minidisk, the output file is written to that directory. If the directory is not writable, the output file is written to your top directory. If the input file is in the XEDIT ring, the output file is written to the minidisk accessed as filemode A.

Note: If you have an existing file with the same name as the C output file, it is overwritten and then erased. Therefore, you may want to use the **savec** option to specify a different name for the output file, as discussed in the following section. △

Saving the output file under CMS

You can use three options to save the intermediate code produced by the translator: **pponly**, **tronly**, and **savec**. Here is a brief description of each of these options:

pponly performs only the preprocessing step. The program is not translated or compiled. The output file contains C++ code.

tronly performs the preprocessing and translation steps. The program is not compiled. The output file contains C code.

savec saves the C code resulting from translation. The program is preprocessed, translated, and compiled. The output file contains C code.

Under CMS, the **savec** option must be the last option on the command line. LCXX uses the remainder of the command line as the fileid of the output file.

For detailed information on these three options, refer to "Option Descriptions" on page 38.

Note: Because both **tronly** and **savec** save the intermediate C code file, you should use only one of them. For example, if you use **tronly**, you do not need to also specify **savec**, unless you want to rename the intermediate C code file. △

Specifying a fileid

The following sections illustrate specifying fileids (for both input and output files). The examples are divided according to your file environment, such as files on accessed minidisk or directory, files on the XEDIT ring, and files in a Shared File System (SFS) directory.

Files on an accessed minidisk or directory

To specify a file on an accessed minidisk or directory, use

```
filename <filetype <filemode>>
```

Optionally, you can separate the filename, filetype, and filemode by periods instead of blanks. If the file is on an accessed minidisk or directory, you can optionally prefix the fileid with **cms:**. If the filetype is omitted, the default filetype for the input file is CXX. If the filemode is omitted, the translator searches all accessed minidisks and directories. Here are some examples:

LCXX MYPROG
: The input file is MYPROG CXX *. The output file is temporary and is named MYPROG TROUT. The output file is written to either the minidisk where the input file was found or to the A disk if the input minidisk is not writable.

LCXX MYPROG.CPP
: The input file is MYPROG CPP *. The output file is temporary and is named MYPROG TROUT.

LCXX MYPROG (SAVEC MYPROG CPPOUT B2
: The input file is MYPROG CXX *. The **savec** option stores the output file as MYPROG CPPOUT B2.

LCXX CMS:MYPROG.CPP.J (NOWARN SAVEC CMS:MYPROG.CPPOUT.B2
: The input file is MYPROG CPP J. The **nowarn** option suppresses translator warning messages. The **savec** option stores the output file as MYPROG CPPOUT B2. Note that the **savec** option must be the last option specified.

Files in the XEDIT ring

If you run the LCXX EXEC from the XEDIT command line and the input file is in the XEDIT ring, the translator automatically reads the input file from XEDIT. However, the translator does not write its output to an XEDIT file.

Files in an SFS directory

To specify a file in an SFS directory, use the following format:

sf: *filename* <*filetype* <*directory-name* | *namedef*>>

The **sf:** prefix is required. If the filetype is omitted, the default filetype is CXX. If you do not specify either a directory-name or a NAMEDEF, the default is your top directory. Here are some examples:

LCXX SF:MYPROG

MYPROG CXX
: is the input file.

MYPROG TROUT
: is the temporary output file.

LCXX SF:MYPROG CPP

MYPROG CPP.

MYPROG TROUT
: is the temporary output file.

LCXX SF:MYPROG (SAVEC MYPROG CPPOUT

MYPROG CXX.
: is the input file.

MYPROG CPPOUT.
: is the permanent output file, created by the **savec** option. Note that the output file prefix defaults to the input file prefix.

LCXX SF:MYPROG CXX CXXPROG

MYPROG CXX CXXPROG
: is the input file, where CXXPROG is a NAMEDEF defined in a NAMEDEF command.

MYPROG TROUT.
: is the temporary output file.

LCXX SF:MYPROG CXX .CXX.PROJ (SAVEC CMS:MYPROG TROUT B2

MYPROG CXX .CXX.PROJ
: is the input file.

MYPROG TROUT B2
: is the permanent output file, created by the **savec** option. The **cms:** prefix is used to override the default prefix of **sf:**.

Locating header files under CMS

The translator searches for header files in one of two locations, depending on how the filename is specified in the **#include** statement. When the filename is surrounded by angle brackets in the **#include** statement, the translator assumes that the header file is a member of a macro library that has been designated GLOBAL. The syntax for this form of the **#include** statement is

```
#include <member.ext>
```

The *.ext* part can be omitted.
For example:

```
#include <new.h>
```

In this case, the translator includes the first member named NEW that it finds in the GLOBAL macro libraries.

When the filename is surrounded by double quotes in the **#include** statement, the translator searches for the specified file. The syntax for this form of the **#include** statement is

```
#include "fileid"
```

fileid can be any CMS fileid in the **cms:**, **xed:**, or **sf:** formats. For example, suppose you had the following **#include** statement in your program:

```
#include "project.h"
```

In this case, the translator attempts to include the file named PROJECT H *. If this file cannot be found, the translator searches the macro libraries that have been designated GLOBAL. The translator uses that part of the fileid preceding any blank or period as the member name. Refer to the *SAS/C Library Reference, Third Edition, Volume 1* for more information about CMS filename formats.

If you are using the translator in a version of CMS that supports the Shared File System, the translator automatically searches your top directory for header files before searching any accessed minidisks. You can specify additional directories to be searched by using the _HEADERS environment variable. Refer to the *SAS/C Compiler and Library User's Guide* for more information about using environment variables.

Before invoking the LCXX EXEC, be sure that the C++ standard header files and C standard header files are available. The C++ standard header files are in a macro library named LCXX370 MACLIB. The C standard header files are in a macro library named LC370 MACLIB. To make these header files available to the translator, issue the following command before calling the translator:

```
GLOBAL MACLIB LCXX370 LC370
```

If your program requires header files in another macro library, add the filename of the macro library to the GLOBAL MACLIB command.

CXXMACLIBS and CXXOPTIONS GLOBALV variables

The LCXX EXEC automatically issues a GLOBAL MACLIB command for you if you set the CXXMACLIBS environment variable to the names of the macro libraries you want to be made available. In addition, if you set the CXXOPTIONS environment variable to a list of translator options, LCXX automatically passes those options to the translator in addition to the options you specify on the command line. Set these variables using the GLOBALV command. Both variables should be in the LC370 group.

The GLOBALV command (see Example Code 2.1 on page 18) sets the CXXMACLIBS variable so that the LCXX EXEC automatically issues a GLOBAL MACLIB command for the C++ standard header files and the C standard header files.

You can suppress the use of environment variables for a single compilation by specifying the NOGLOBAL option on the LCXX command line.

The following example shows how to set the CXXOPTIONS variable to cause LCXX to automatically pass the **overload** option to the translator:

```
GLOBALV SELECT LC370 SETP CXXOPTIONS OVERLOAD
```

Options specified on the command line override options specified via the CXXOPTIONS variable.

Linking Your Program under CMS

All C++ programs must be preprocessed by the COOL object code preprocessor because COOL automatically creates object code used during the initialization and termination of static objects. In addition, COOL supports linking object files that contain mixed case external names longer than eight characters, such as those created by the compiler for source code generated by the translator. Such external names cannot be handled by the linkage editor unless the object code has been preprocessed by COOL.

COOL EXEC

The COOL EXEC calls COOL to preprocess your object code, and it optionally invokes the CMS GENMOD command. The format of the COOL EXEC is

COOL <*filename1* <*filename2* . . .>> (CXX <*options*<)>>

Filename1, *filename2*, and so on are the filenames of primary input files or AR370 archives. For each filename in the list, COOL first checks for an AR370 archive with this filename; if one is not found, it looks for a TEXT file. For instance, if the filename FINANCE is specified, COOL first looks for FINANCE A, an AR370 archive on an accessed minidisk or in an SFS directory accessed as a minidisk. If FINANCE A is not found, COOL looks for FINANCE TEXT. Any TEXT files specified on the command line may contain object code, COOL control statements, or both. Any AR370 archives specified on the command line are used to resolve unresolved references during the processing of the other input files.

For example, the following command line causes COOL to use MYPROG TEXT as its only input file, provided that MYPROG A is not found:

```
COOL MYPROG (CXX
```

If you do not specify any filenames, COOL prompts you for filenames. Enter as many filenames as necessary in response to the COOL: prompt. Enter a null string (that is, press the ENTER key) to cause COOL to begin processing the input files.

The CXX option is the only required option. This option causes the standard C++ object library to be added to COOL's autocall list. Refer to "COOL Options" on page 30 for more information about COOL options. Before calling COOL, you must issue a GLOBAL TXTLIB command to make the standard C object libraries available for autocall resolution. The standard C object libraries are LC370BAS TXTLIB and LC370STD TXTLIB.* The following GLOBAL TXTLIB command makes these libraries available for COOL:

```
GLOBAL TXTLIB LC370BAS LC370STD
```

* The standard C++ object library is named LIBCXX A. Other run-time libraries include LC370GOS TXTLIB for use with the Generalized Operating System, LC370SPE TXTLIB for use with the Systems Programming Environment, and LC370CIC TXTLIB for use with the SAS/C CICS Command Language Translator.

Example Code 2.1 CXXMACLIBS GLOBALV Command

```
GLOBALV SELECT LC370 SETLP CXXMACLIBS LCXX370
LC370
```

The COOL EXEC automatically issues a GLOBAL TXTLIB command for you if you specify the names of the object libraries in the TXTLIBS environment variable.

The COOL input files can contain control statements instead of, or in addition to, object code. For more information about COOL control statements, refer to "COOL Control Statements" on page 29.

COOL writes the preprocessed object code to a file named COOL370 TEXT A1. This file can be used as input to the LOAD command or the LKED command.

The COOL listing file is written to the terminal by default. You can redirect it to a disk file. Enter the redirection argument in the filename part of the command line, not the options part. For example, the following command causes COOL to write its listing file to MYPROG COOLMAP A1:

```
COOL MYPROG >MYPROG.COOLMAP.A1 (CXX
```

There are many special considerations for linking C++ programs under CMS that parallel the considerations for linking C programs. Refer to the *SAS/C Compiler and Library User's Guide* for a complete discussion of these considerations.

Creating a MODULE file

The COOL EXEC accepts the GENMOD option, which specifies that the EXEC should load the COOL370 TEXT file and issue the GENMOD command. You can specify the name of the MODULE file following the GENMOD option. If you do not specify a name, the COOL EXEC uses the first input filename on the command line as the name of the MODULE. For example, the following command creates MYPROG MODULE:

```
COOL MYPROG LIBFNC (CXX GENMOD
```

The following command creates APPL1 MODULE:

```
COOL MAIN1 SUB1 SUB2 (CXX GENMOD APPL1
```

You can also cause COOL to invoke the START command or the LKED command. Refer to the *SAS/C Compiler and Library User's Guide* for more information.

Running Your Program under CMS

C++ programs can be run just like any other program under CMS. The most frequently used method for running a program is to create a MODULE file (as shown previously) and then call the module as a CMS command. For example, the following command calls MYPROG MODULE, passing the program option **-z**, the run-time option **=48k**, and redirecting **stdin** to INPUT FILE *:

```
MYPROG -z =48k <INPUT.FILE
```

You can also load a TEXT file and use the START command to run it. For example, suppose you have created MYFILE TEXT by using COOL to preprocess MYPROG TEXT. The following commands can be used to load and run it, passing the same options as in the previous example:

```
LOAD MYFILE
START * -z =48k <INPUT.FILE
```

Programs that can be called from the CMS EXEC processor should be prepared to accept tokenized parameters. This form of parameter is at most eight characters long and is translated to uppercase by CMS. Refer to the *SAS/C Compiler and Library User's Guide* for more information. If you want to run your program with the SAS/C Debugger, use the '=D' option when you call your program, as in the following example:

```
MYPROG =D
```

For more information on debugging C++ programs, refer to Chapter 5, "Debugging C++ Programs Using the SAS/C Debugger," on page 97.

Creating C++ Programs under MVS Batch

This section describes six cataloged procedures that you can use to translate, compile, link, and run a C++ program. Table 2.1 on page 19 lists the procedures.

Table 2.1 C++ Cataloged Procedures

Procedure	Translate and Compile	COOL and Link Edit	Run	Store AR370 Archive Member
LCXXC	Yes	No	No	No
LCXXCA	Yes	No	No	Yes
LCXXCL	Yes	Yes	No	No
LCXXL	No	Yes	No	No
LCXXCLG	Yes	Yes	Yes	No
LCXXLG	No	Yes	Yes	No

Each subsequent section describes using the procedures, giving both the general syntax and an example.

General Notes about the Cataloged Procedures

Note the following considerations when you use any of the cataloged procedures described in this section:

- The actual cataloged procedures may differ slightly from the versions shown here due to changes since this book was written.
- If you override SYSPRINT to describe a disk data set, the data set disposition must be MOD. A SYSPRINT data set cannot be a member of a PDS.
- The SYSLIB parameter refers to the site-selected name for an autocall library. Do not override this parameter.
- The CALLLIB parameter can be used to specify an object module call library to be used in addition to the C standard object library.

  In general, the CALLLIB parameter should not be used for a library containing C++ object modules. This is because MVS partitioned data sets are not suitable for storing C++ object modules, due to the limitation of eight-character uppercase member names. You should store C++ object code in AR370 archives and access them for autocall via the SYSARLIB DD statement.

Translating and Compiling Your Program under MVS Batch

You can use one of two cataloged procedures to simply translate and compile your C++ program and do nothing else (no link or run step). The LCXXC cataloged procedure invokes the translator and compiler in one job step and stores the compiler output in an MVS data set. The LCXXCA cataloged procedure is identical to the LCXXC procedure, except that the compiler output is stored in an AR370 archive. You can optionally run the OMD370 utility with either of these cataloged procedures.

If you only want to translate (not compile, link, or run) your C++ program, use the LCXXC cataloged procedure and use the **`tronly`** option. This option causes only the translator to be run. Similarly, if you only want to preprocess your code, use the LCXXC cataloged procedure with the pponly option. For more information about options you can use the with the translator, refer to "Option Summary" on page 35.

DD statements used in translation and compilation

The cataloged procedures consist of various JCL statements, some of which are DD statements. Sometimes you may want to override some of the DD statements in the procedures. The following list describes the most commonly overridden DD statements in the translation and compilation steps. The descriptions include a brief overview of each statement's purpose and any pertinent DCB requirements for the data sets described by the DD statement.

Note: These DD statements apply to not only the LCXXC and LCXXCA procedures but to all cataloged procedures that translate and compile a program (LCXXC, LCXXCA, LCXXCL, and LCXXCLG). △

SYSTROUT DD statement
: describes the data set that is used to contain the translator output. Usually this is a temporary data set. The data set can be sequential or a member of a PDS.

SYSTRIN DD statement
: describes the translator input data set. This data set can have either fixed- or variable-length records of any length. It can be blocked or unblocked and can be sequential or a member of a PDS. The SYSTRIN DD statement is required.

SYSLIB DD statement
: describes one or more standard header file data sets. These data sets must be partitioned. They can have either fixed- or variable-length records of any length and can be blocked or unblocked. However, all SYSLIB data sets must have the same record format and record length, and the one with the largest block size must be first. Usually, SYSLIB includes SASC.MACLIBC; if so, all other partitioned data sets must match the SASC.MACLIBC record format and record length (RECFM=FB, LRECL=80). The SYSLIB DD statement is required if any standard header files are included in the primary input file.

SYSLIN
: describes a file that is used for the compiled object code. It can be a sequential data set or a PDS member. Its DCB should specify RECFM=FB and LRECL=80, and the BLKSIZE value should be no greater than 3200. No SYSLIN statement should be provided if you use the LCXXCA cataloged procedure.

SYSDBLIB
: describes a file that contains debugging information for the compilation. This file must be a PDS. Its DCB should specify RECFM=U and BLKSIZE=4080. A SYSDBLIB DD statement is needed only if you use the translator debug option, DBGOBJ, or use the AUTOINST option.

SYSARLIB
: applies only to the LCXXA cataloged procedure. It describes an output AR370 archive. The compiled object code is stored in a member of the archive. Because C++ programs use extended names, if you want to autocall your C++ functions, you must store them in an AR370 archive, not in a PDS. For more information on AR370 archives, refer to the *SAS/C Compiler and Library User's Guide*.

In addition, the translator may require one or more DD statements describing user-defined header files, as mentioned previously.

LCXXC cataloged procedure

In general, the LCXXC cataloged procedure is used as shown here:

```
//SAMPLE JOB jobcard information
//*
//STEP1 EXEC LCXXC,PARM.X='options'
//X.SYSLIN DD DISP=OLD,DSN=your.object.dataset
//X.SYSTRIN DD DISP=SHR,DSN=your.source.dataset
//X.ext DD DISP=SHR,DSN=your.headers.dataset
//
```

options are any translator, compiler, or, if the OMD option is used to run the OMD370 utility, OMD370 options. See "Option Summary" on page 35 for the options that are available for use with the translator. You need only provide a SYSTRIN DD statement to describe your source data set and a SYSLIN DD statement to describe your object data set. The X.*ext* DD statement is optional and describes the PDS that the translator searches for header files. Example Code 2.2 on page 20 illustrates using the LCXXC cataloged procedure.

Example Code 2.2 Using the LCXXC Cataloged Procedure

```
//PROJECT1 JOB jobcard information
```

```
//*
//* Translator input: FRIEND.PROJ.CXX(MAIN1)
//* Translator options: NONE
//* Compiler options:   RENT
//* Header Files:       FRIEND.PROJ.H,
                        LEADER.SYSTEM.H
//* Object code:      FRIEND.PROJ.OBJ(MAIN1)
//*
//MAINXC    EXEC LCXXC,PARM.X='RENT'
//X.SYSLIN  DD DISP=OLD,
//          DSN=FRIEND.PROJ.OBJ(MAIN1)
//X.SYSTRIN DD DISP=SHR,
//          DSN=FRIEND.PROJ.CXX(MAIN1)
//X.H       DD DISP=SHR,DSN=FRIEND.PROJ.H
//          DD DISP=SHR,DSN=LEADER.SYSTEM.H
//
```

LCXXCA cataloged procedure

In general, the LCXXCA cataloged procedure is used as shown here:

```
//SAMPLE JOB jobcard information
//*
// EXEC LCXXCA,MEMBER=AR-archive-member,
//      PARM='options'
//X.SYSTRIN DD DISP=SHR,
//          DSN=your.source.dataset
//A.SYSARLIB DD DISP=OLD,
//          DSN=your.AR.dataset
//
```

Note that the MEMBER= parameter is required with this cataloged procedure. *AR-archive-member* is the AR370 archive member in which you want the compiler output stored. The **options** are any translator, compiler, or, if the OMD option is used to run the OMD370 utility, OMD370 options. See "Option Summary" on page 35 for the options that are available for use with the translator. You need only provide a SYSTRIN DD statement to describe your source data set and a SYSARLIB DD statement to describe your output AR370 archive. The X.*ext* DD statement is optional and describes the PDS that the translator searches for header files. Note that you never use a SYSLIN DD statement with the LCXXCA cataloged procedure. Example Code 2.3 on page 21 illustrates using the LCXXCA cataloged procedure.

Example Code 2.3 Using the LCXXCA Cataloged Procedure

```
//PROJECT1 JOB jobcard information
//*
//* Translator input:  FRIEND.PROJ.CXX(MAIN1)
//* Translator options: SNAME
//* Compiler options:   RENT
//* AR370 archive/member: PROJECT.CXX.A(PARSER)
//*
// EXEC LCXXCA,MEMBER=PARSER,
//             PARM='RENT,SNAME(PARSER)'
//X.SYSTRIN DD DISP=SHR,
//          DSN=FRIEND.PROJ.CXX(MAIN1)
//A.SYSARLIB DD DISP=OLD,DSN=PROJECT.CXX.A
//
```

Saving the output data set under MVS batch

Under MVS batch, the intermediate code produced by the translator is sent to the DDname SYSTROUT, which by default is associated with a temporary file. You can make the output file permanent by using the **pponly** or **tronly** options in combination with altering the SYSTROUT DD statement. Here is a brief description of each of these options:

pponly performs only the preprocessing step. The program is not translated or compiled. The output data set contains C++ code.

tronly performs the preprocessing and translation steps. The program is not compiled. The output data set contains C code.

To save the C code resulting from translation when compiling your program, change SYSTROUT to refer to a permanent data set (no option is necessary). For detailed information on the **pponly** and **tronly** options, refer to "Option Descriptions" on page 38.

Locating header files under MVS batch

When the filename is surrounded by angle brackets in an **#include** statement, the translator searches for the header file in standard C and C++ header files supplied by SAS Institute. The format of this form of the **#include** statement is:

```
#include <member.ext>
```

The *.ext* part can be omitted. Here is an example of including a standard header file:

```
#include <new.h>
```

For this statement, the translator includes the file named ddn:SYSLIB(NEW). To include a header file from your personal files (as opposed to header files from system files), surround the filename with double quotes, as shown here:

```
#include "member.ext"
```

The translator assumes that the header file is *member* in a PDS described by the *ext* DD statement. The *.ext* part can be omitted, in which case it defaults to *.h*. For example, suppose you had the following **#include** statement in your program:

```
#include "project.h"
```

In this case, the translator attempts to include the member PROJECT from the data set associated with the DDname H. If the file is not found, the translator searches the system header files for the member.

Linking Your Program under MVS Batch

All C++ programs must be preprocessed by the COOL object code preprocessor because COOL automatically creates object code used during the initialization and termination of static objects. In addition, COOL supports linking object files that contain mixed case external names longer than eight characters, such as those created by the compiler for source code generated by the translator. Such external names cannot be handled by the linkage editor unless the object code has been preprocessed by COOL. This section describes the LCXXCL and LCXXL cataloged procedures.

DD statements used in linking

The cataloged procedures consist of various JCL statements, some of which are DD statements. Sometimes you may want to override some of the DD statements in the procedures. The following list describes the most commonly overridden DD statements in the linking step. The descriptions include a brief overview of each statement's purpose and any pertinent DCB requirements for the data sets described by the DD statement.

Note: These DD statements apply to not only the LCXXCL and LCXXL procedures but to all cataloged procedures that link programs (LCXXCL, LCXXL, LCXXLG, and LCXXCLG). △

SYSDBLIB
: describes a file that contains debugging information for the compilation. This file must be a PDS. Its DCB should specify RECFM=U and BLKSIZE=4080. A SYSDBLIB DD statement is needed only if you use the debug option, DBGOBJ, or use the AUTOINST option.

SYSLIB
: defines an object code call library to COOL. If you want to use a single additional call library, you should use the CALLLIB symbolic parameter, as described in the *SAS/C Compiler and Library User's Guide*. If you need several such libraries, you should concatenate them after the standard libraries referenced by the cataloged procedure.

SYSLDLIB
: is an optional DD statement that defines any user or system autocall libraries needed in load module form. Members in SYSLDLIB are left unresolved by COOL and are resolved by the linkage editor.

SYSLMOD
: defines the load module library and the member where the output of the linkage editor is to be stored.

SYSIN
: identifies the primary input to COOL. If you are using a procedure that runs the translator and compiler as well as COOL, you should not define your own SYSIN DD statement, as one is defined automatically by the procedure. If you are using LCXXL or LCXX LG, you must specify SYSIN. The SYSIN data set can be a sequential data set or a PDS member, and may contain object code, link-edit control statements such as INCLUDE, or both.

SYSARLIB
: identifies one or more input AR370 archives. Each AR370 archive contains C or C++ routines to be linked into a load module by autocall. Because C++ programs use extended names, if you want to autocall your C++ functions you must store them in an AR370 archive, not in a PDS. For more information on AR370 archives, refer to the *SAS/C Compiler and Library User's Guide*.

If you want to concatenate your own AR370 archive to the C++ library archive, you can do so. Here is an example of such a statement, which puts your AR370 archive (PROJECT.CPLUS.A) after the library archive:

```
//LKED.SYSARLIB DD
//   DD DSN=PROJECT.CPLUS.A,DISP=SHR
```

Alternatively, you can put your archive first:

```
//LKED.SYSARLIB DD DSN=PROJECT.CPLUS.A,
//   DISP=SHR
//   DD DSN=SASC.CXX.A,DISP=SHR
```

The SYSARLIB archives must have RECFM=U.

In addition, one or more DD statements describing user INCLUDE libraries or AR370 archives may be required.

ENV and ALLRES parameters

The LCXXCL and LCXXL cataloged procedures, as well as the LCXXCLG and LCXXLG cataloged procedures described in the next section, support the ENV parameter and the ALLRES parameter.

The ENV parameter is used to select the program environment. The default is ENV=STD, indicating that the module runs in a normal C environment.

The ALLRES parameter is used to specify whether the program uses the all-resident library or the transient library. The default is ALLRES=NO.

In most cases, these parameters should be left to take their default values. Refer to the *SAS/C Compiler and Library User's Guide* for more information about these parameters.

LCXXCL cataloged procedure

The LCXXCL cataloged procedure invokes the translator and compiler in one job step, followed by COOL and the linkage editor in another step. As in the LCXX procedure, you can also optionally run the OMD370 utility. In general, the LCXXCL cataloged procedure is used as shown in Example Code 2.4 on page 22.

Example Code 2.4 LCXXCL Cataloged Procedure

```
//SAMPLE JOB    jobcard information
//*
//STEP1 EXEC LCXXCL,CALLLIB='your.object.lib',
//       PARM.X='C++-options',
//       PARM.LKED='COOL-options'
//X.SYSTRIN  DD DISP=SHR,DSN=your.source.dataset
//X.ext DD DISP=SHR,DSN=your.headers.dataset
//LKED.SYSARLIB DD
//       DD DISP=SHR,DSN=private.AR370.archive
//LKED.SYSLMOD  DD DISP=OLD,
//              DSN=your.load.module(member)
//LKED.SYSLDLIB DD DSN=your-autocall-load-lib,
//              DISP=SHR
//NAME DD DISP=SHR
//     DD DSN=OTHER.NAME
```

By default, the LCXXCL cataloged procedure passes the LIST and MAP options to the linkage editor. You can override this default by specifying different options via PARM.LKED. You should provide a SYSTRIN DD statement to describe your source data set and a SYSLMOD DD statement to describe your load module data set. Example Code 2.5 on page 23 illustrates using the LCXXCL cataloged procedure.

Example Code 2.5 Using the LCXXCL Cataloged Procedure

```
//PROJECT1 JOB jobcard information
//*
//* Translator input:  FRIEND.PROJ.CXX(MAIN1)
//* Translator options:  NONE
//* Compiler options:    RENT
//* Header files:        FRIEND.PROJ.H,
//*                      LEADER.SYSTEM.H
//* Autocall object library: LEADER.SYSTEM.OBJ
//* Linkage Editor options: LIST,MAP,XREF,
//*                         AMODE=31,RMODE=ANY
//* AR370 archive:          PROJECT.CPLUS.A
//* Load module:      FRIEND.PROJ.LOAD(MAIN1)
//* Autocall load library:  SYS1.ISPLOAD
//* (ISPF interface load module)
//*
//MAINXCL     EXEC LCXXCL,
//            PARM.X='RENT',
//            PARM.LKED=('LIST,MAP,XREF',
//            'AMODE=31,RMODE=ANY'),
//            CALLLIB='LEADER.SYSTEM.OBJ'
//X.SYSTRIN   DD DISP=SHR,
//            DSN=FRIEND.PROJ.CXX(MAIN1)
//X.H         DD DISP=SHR,DSN=FRIEND.PROJ.H
//            DD DISP=SHR,DSN=LEADER.SYSTEM.H
//LKED.SYSARLIB DD
//            DD DISP=SHR,DSN=PROJECT.CPLUS.A
//LKED.SYSLMOD  DD DISP=OLD,
//              DSN=FRIEND.PROJ.LOAD(MAIN1)
//LKED.SYSLDLIB DD DISP=SHR,DSN=SYS1.ISPLOAD
//
```

LCXXL cataloged procedure

The LCXXL cataloged procedure invokes COOL and the linkage editor to preprocess and linkedit object code produced by another job. In general, the LCXXL cataloged procedure is used as shown in Example Code 2.6 on page 23.

Example Code 2.6 LCXXL Cataloged Procedure

```
//SAMPLE JOB    jobcard information
//*
//STEP1 EXEC LCXXL,
//      CALLLIB='your.object.library',
//      PARM.LKED='options'
//LKED.SYSARLIB DD
//      DD DISP=SHR,
//      DSN=private.AR370.archive
//LKED.SYSLMOD  DD DISP=OLD,
//          DSN=your.load.module(member)
//LKED.SYSIN  DD DISP=SHR,
//            DSN=your.object.dataset
//LKED.libname DD DISP=SHR,
//            DSN=your.object.library
//LKED.SYSLDLIB DD DSN=your-autocall-load-lib,
//             DISP=SHR
//
```

By default, the LCXXL cataloged procedure passes the LIST and MAP options to the linkage editor. You can override this default by specifying different options via PARM.LKED. You should provide a SYSIN DD statement to describe your primary input data set, and a SYSLMOD DD statement to describe your load module data set. Example Code 2.7 on page 23 illustrates using the LCXXL cataloged procedure.

Example Code 2.7 Using the LCXXL Cataloged Procedure

```
//PROJECT1 JOB   jobcard information
//*
//* Linkage Editor options: LIST,MAP,XREF,
//*                         AMODE=31,RMODE=ANY
//* COOL primary input:
//*                FRIEND.PROJ.OBJ(MAIN1)
//* COOL secondary input: LEADER.SUBS.OBJ
//* Autocall object library:
//*                LEADER.SYSTEM.OBJ
//* Load module:  FRIEND.PROG.LOAD(MAIN1)
//* AR370 archive:       PROJECT.CPLUS.A
//* Autocall load library: SYS1.ISPLOAD
//*
//MAINXL        EXEC LCXXL,
//              PARM.LKED=('LIST,MAP,XREF',
//              'AMODE=31,RMODE=ANY'),
//              CALLLIB='LEADER.SYSTEM.OBJ'
//LKED.SYSARLIB DD
//          DD DISP=SHR,DSN=PROJECT.CPLUS.A
//LKED.SYSLMOD  DD DISP=OLD,
//              DSN=FRIEND.PROJ.LOAD(MAIN1)
//LKED.SYSIN    DD DISP=SHR,
//              DSN=FRIEND.PROJ.OBJ(MAIN1)
//LKED.SUBLIB DD DISP=SHR,
//              DSN=LEADER.SUBS.OBJ
//LKED.SYSLDLIB DD DISP=SHR,DSN=SYS1.ISPLOAD
//
```

Running Your Program under MVS Batch

You can choose to use one of two cataloged procedures to run your program, as described in this section. The LCXXCLG procedure translates, compiles, links, and runs a program. The LCXXLG procedure simply links and runs a program.

DD statements used at run time

To run a C++ program, the DD statements described in the following list may be required.

SYSTERM
: describes the data set to which the **cerr** stream should be written. The usual specification for this statement is SYSOUT=A. This DD statement is optional but highly recommended. All library error messages are written to this file.

SYSPRINT
: describes the data set to which the cout stream should be written. This DD statement is optional.

SYSIN
: describes the data set from which the **cin** stream is read. This DD statement is optional.

Other DD statements may be needed if you want to run your program under the SAS/C Debugger. Refer to the *SAS/C Debugger User's Guide and Reference* for more information. In addition, further DD statements may be required to define files opened by your program.

Specify program arguments, library options, standard file redirections, and environment variables via the PARM.GO parameter of the EXEC statement. If you want to generate a dump, use a SYSUDUMP DD statement.

LCXXCLG cataloged procedure

The LCXXCLG cataloged procedure is identical to the LCXXCL cataloged procedure, with the addition of a GO step to run the program. In general, the LCXXCLG cataloged procedure is used as shown in Example Code 2.8 on page 24.

Example Code 2.8 LCXXCLG Cataloged Procedure

```
//SAMPLE JOB    jobcard information
//*
//STEP1         EXEC LCXXCLG,
//              CALLLIB='your.object.library',
//              PARM.X='C++-options',
//              PARM.LKED='COOL-options',
//              PARM.GO='program-options'
//X.SYSTRIN     DD DISP=SHR,
//              DSN=your.source.dataset
//X.ext         DD DISP=SHR,
//              DSN=your.headers.dataset
//LKED.SYSARLIB DD DD DISP=SHR,
//               DSN=private.AR370.archive
//LKED.libname  DD DISP=SHR,
//               DSN=your.object.library
//LKED.SYSLDLIB DD DSN=your-autocall-load-lib,
//               DISP=SHR
//GO.DBGIN      DD DISP=SHR,
//               DSN=your.debugger.input
//GO.SYSIN      DD DISP=SHR,
//               DSN=your.program.input
//
```

Pass run-time options for your program such as program arguments, library options, and standard file redirections via the PARM.GO parameter. In addition to the DD statements used with the LCXXCL cataloged procedure, you should provide a SYSIN DD statement for the GO step to describe the **cin** input stream for your program, if necessary. By default, the **cout** stream (SYSPRINT) and **cerr** stream (SYSTERM) are specified as SYSOUT data sets. Example Code 2.9 on page 24 illustrates using the LCXXCLG cataloged procedure.

Example Code 2.9 Using the LCXXCLG Cataloged Procedure

```
//PROJECT1 JOB      jobcard information
//*
//* Translator input:  FRIEND.PROJ.CXX(MAIN1)
//* Translator options:  NONE
//* Compiler options:    NONE
//* Header files:        FRIEND.PROJ.H,
//*                      LEADER.SYSTEM.H
//* Autocall object library: LEADER.SYSTEM.OBJ
//* Linkage Editor options: LIST,MAP,XREF,
//*                         AMODE=31,RMODE=ANY
//* AR370 archive:          PROJECT.CPLUS.A
//* Program arguments:      =48k -z
//* Program input:     MY.SAMPLE.INPUT(TEST1)
//*
//MAINXCLG    EXEC LCXXCLG,
//            CALLLIB='LEADER.SYSTEM.OBJ'
//            PARM.LKED=('LIST,MAP,XREF',
//            'AMODE=31,RMODE=ANY'),
//            PARM.GO='=48k -z'
//X.SYSTRIN   DD DISP=SHR,
//            DSN=FRIEND.PROJ.CXX(MAIN1)
//X.H         DD DISP=SHR,DSN=FRIEND.PROJ.H
//            DD DISP=SHR,DSN=LEADER.SYSTEM.H
//LKED.SYSARLIB DD
//            DD DISP=SHR,DSN=PROJECT.CPLUS.A
//GO.SYSIN    DD DISP=OLD,
//            DSN=MY.SAMPLE.INPUT(TEST1)
//
```

LCXXLG cataloged procedure

The LCXXLG cataloged procedure is identical to the LCXXL cataloged procedure described earlier, with the addition of a GO step to run the linkedited program. In general, the LCXXLG cataloged procedure is used as shown in Example Code 2.10 on page 24.

Example Code 2.10 LCXXLG Cataloged Procedure

```
//SAMPLE JOB jobcard information
//*
//STEP1 EXEC LCXXLG,
//      CALLLIB='your.object.library',
//      PARM.LKED='COOL-options',
//      PARM.GO='program-options'
//LKED.SYSIN  DD DISP=SHR,
//            DSN=your.object.dataset
//LKED.libname  DD DISP=SHR,
//              DSN=your.object.library
//LKED.SYSLDLIB DD DSN=your-autocall-load-lib,
//              DISP=SHR
//LKED.SYSARLIB DD DD DISP=SHR,
//              DSN=private.AR370.archive
//GO.DBGIN    DD DISP=SHR,
//            DSN=your.debugger.input
//GO.SYSIN    DD DISP=SHR,
//            DSN=your.program.input
//
```

Pass run-time options for your program such as program arguments, library options, and standard file redirections via the PARM.GO parameter. In addition to the DD statements used with the LCXXL cataloged procedure, you should provide a SYSIN DD statement for the GO step to describe the cin stream for your program, if necessary. By default, the **cout** stream (SYSPRINT) and **cerr** stream (SYSTERM) are specified as SYSOUT data sets. Example Code 2.11 on page 25 illustrates using the LCXXLG cataloged procedure.

Example Code 2.11 Using the LCXXLG Cataloged Procedure

```
//PROJECT1 JOB   jobcard information
//*
//* COOL autocall:      LEADER.SYSTEM.OBJ
//* Linkage Editor options: LIST,MAP,XREF,
//*                      AMODE=31,RMODE=ANY
//* AR370 archive:       PROJECT.CPLUS.A
//* Program arguments:      =48k -z
//* Program input:    MY.SAMPLE.INPUT(TEST1)
//*
//MAINLG       EXEC LCXXLG,
//             CALLLIB='LEADER.SYSTEM.OBJ',
//             PARM.LKED=('LIST,MAP,XREF',
//             'AMODE=31,RMODE=ANY'),
//             PARM.GO='=48k -z'
//LKED.SYSIN   DD DISP=SHR,
//              DSN=FRIEND.PROJ.OBJ(MAIN1)
//LKED.SYSARLIB DD DISP=SHR,
//              DSN=PROJECT.CPLUS.A
//GO.SYSIN      DD DISP=OLD,
//             DSN=MY.SAMPLE.INPUT(TEST1)
//
```

Cataloged Procedure Listings

This section lists the JCL for the six cataloged procedures available for the translator. Use this section for reference to determine the proper order of DD statements when you are writing overriding JCL. The procedures are listed in the following order: LCXXC, LCXXCA, LCXXCL, LCXXL, LCXXCLG, and LCXXLG.

JCL for the LCXXC cataloged procedure

This procedure translates and compiles your code.

```
//LCXXC PROC
//*******************************************
//* NAME: LCXXC
//* SUPPORT: C COMPILER DIVISION
//* PRODUCT: SAS/C C++ DEVELOPMENT SYSTEM
//* PROCEDURE: TRANSLATE AND COMPILE
//* DOCUMENTATION: SAS/C C++ DEVELOPMENT
//*                SYSTEM USER'S GUIDE
//* FROM: SAS INSTITUTE INC., SAS CAMPUS DR.,
//*       CARY, NC 27513
//*******************************************
//*
//X        EXEC PGM=LC370CX
//STEPLIB  DD DSN=SASC.LOAD,
//         DISP=SHR TRANSLATOR LIBRARY
//         DD DSN=SASC.LINKLIB,
//         DISP=SHR RUNTIME LIBRARY
//SYSTERM  DD SYSOUT=A
//SYSPRINT DD SYSOUT=A
//SYSTROUT DD DSN=&&TROUT,
//         SPACE=(6160,(10,10)),
//         DISP=(NEW,PASS),UNIT=SYSDA
//SYSIN    DD DSN=*.SYSTROUT,
//         VOL=REF=*.SYSTROUT,
//         DISP=(OLD,PASS)
//SYSUT1   DD UNIT=SYSDA,SPACE=(TRK,(10,10))
//SYSUT2   DD UNIT=SYSDA,SPACE=(TRK,(10,10))
//SYSUT3   DD UNIT=SYSDA,SPACE=(TRK,(10,10))
//SYSTRDB  DD UNIT=SYSDA,SPACE=(TRK,(10,10))
//SYSLIN   DD DSN=&&OBJECT,
//         SPACE=(3200,(10,10)),
//         DISP=(MOD,PASS),UNIT=SYSDA,
//         DCB=(RECFM=FB,LRECL=80)
//SYSLIB   DD DSN=SASC.MACLIBC,
//      DISP=SHR C++ AND C STANDARD HEADERS
//SYSDBLIB DD DSN=&&DBGLIB,
//     SPACE=(4080,(20,20,1)),DISP=(,PASS),
//     UNIT=SYSDA,DCB=(RECFM=U,BLKSIZE=4080)
//SYSTMP01 DD UNIT=SYSDA,
//         SPACE=(TRK,25) VS1 ONLY
//SYSTMP02 DD UNIT=SYSDA,
//         SPACE=(TRK,25) VS1 ONLY
```

JCL for LCXXCA cataloged procedure

This procedure translates and compiles your code and stores the compiler output (object code) in an AR370 archive.

```
//LCXXCA PROC MEMBER=DO.NOT.OMIT
//*******************************************
//* NAME: LCXXCA
//* SUPPORT: C COMPILER DIVISION
//* PRODUCT: SAS/C C++ DEVELOPMENT SYSTEM
//* PROCEDURE: TRANSLATE AND COMPILE
//* DOCUMENTATION: SAS/C C++ DEVELOPMENT
//*                SYSTEM USER'S GUIDE
//* FROM: SAS INSTITUTE INC., SAS CAMPUS DR.,
//*       CARY, NC 27513
//*******************************************
//*
//X        EXEC PGM=LC370CX
//STEPLIB  DD DSN=SASC.LOAD,
//         DISP=SHR TRANSLATOR LIBRARY
//         DD DSN=SASC.LINKLIB,
//         DISP=SHR RUNTIME LIBRARY
//SYSTERM  DD SYSOUT=A
//SYSPRINT DD SYSOUT=A
//SYSTROUT DD DSN=&&TROUT,
//         SPACE=(6160,(10,10)),
//         DISP=(NEW,PASS),UNIT=SYSDA
```

```
//SYSIN    DD DSN=*.SYSTROUT,
//         VOL=REF=*.SYSTROUT,
//         DISP=(OLD,PASS)
//SYSUT1   DD UNIT=SYSDA,
//         SPACE=(TRK,(10,10))
//SYSUT2   DD UNIT=SYSDA,
//         SPACE=(TRK,(10,10))
//SYSUT3   DD UNIT=SYSDA,
//         SPACE=(TRK,(10,10))
//SYSTRDB  DD UNIT=SYSDA,
//         SPACE=(TRK,(10,10))
//SYSLIN   DD DSN=&&OBJECT(&MEMBER),
//         SPACE=(3200,(10,10,1)),
//         DISP=(NEW,PASS),UNIT=SYSDA,
//      DCB=(RECFM=FB,LRECL=80,DSORG=PO)
//SYSLIB   DD DSN=SASC.MACLIBC,
//    DISP=SHR C++ AND C STANDARD HEADERS
//SYSDBLIB DD DSN=&&DBGLIB,
//         SPACE=(4080,(20,20,1)),
//         DISP=(,PASS),
//         UNIT=SYSDA,
//         DCB=(RECFM=U,BLKSIZE=4080)
//SYSTMP01 DD UNIT=SYSDA,
//         SPACE=(TRK,25) VS1 ONLY
//SYSTMP02 DD UNIT=SYSDA,
//         SPACE=(TRK,25) VS1 ONLY
//A        EXEC PGM=AR370#,PARM=R,
//         COND=(4,LT,X)
//STEPLIB  DD DSN=SASC.LOAD,
//         DISP=SHR COMPILER LIBRARY
//         DD DSN=SASC.LINKLIB,
//         DISP=SHR RUNTIME LIBRARY
//SYSTERM  DD SYSOUT=A
//SYSPRINT DD SYSOUT=A
//SYSARLIB DD DSN=&&AR,
//         SPACE=(4080,(10,10)),
//         DISP-(NEW,PASS),UNIT=SYSDA
//OBJECT   DD DSN=*.C.SYSLIN,
//         VOL=REF=*.C.SYSLIN,
//         DISP=(OLD,PASS)
//SYSIN    DD DSN=SASC.BASEOBJ(AR@OBJ),
//         DISP=SHR
```

JCL for the LCXXCL cataloged procedure

This procedure translates, compiles, and links your program.

```
//LCXXCL PROC ENV=STD,ALLRES=NO,
//       CALLLIB='SASC.BASEOBJ',
//       MACLIB='SASC.MACLIBC',
//       SYSLIB='SASC.BASEOBJ',
//       CXXLIB='SASC.LIBCXX.A'
//*****************************************
//* NAME: LCXXCL
//* SUPPORT: C COMPILER DIVISION
//* PRODUCT: SAS/C C++ DEVELOPMENT SYSTEM
//* PROCEDURE: TRANSLATE, COMPILE, COOL,
//*            LINK EDIT
//* DOCUMENTATION: SAS/C C++ DEVELOPMENT
//*                SYSTEM USER'S GUIDE
//* FROM: SAS INSTITUTE INC., SAS CAMPUS DR.,
//*       CARY, NC 27513
//*****************************************
//*
//*****************************************
//* ENV=STD: MODULE RUNS IN THE NORMAL C
//*          ENVIRONMENT
//* ENV=CICS: MODULE RUNS IN A
//*           CICS C ENVIRONMENT
//* ENV=GOS: MODULE RUNS USING THE
//*          GENERALIZED SYSTEM ENVIRONMENT
//* ENV=SPE: MODULE USES THE SYSTEMS
//*          PROGRAMMING ENVIRONMENT
//*****************************************
//X        EXEC PGM=LC370CX
//STEPLIB  DD DSN=SASC.LINKLIB,
//         DISP=SHR C RUNTIME LIBRARY
//         DD DSN=SASC.LOAD,
//         DISP=SHR TRANSLATOR LIBRARY
//SYSTERM  DD SYSOUT=A
//SYSPRINT DD SYSOUT=A
//SYSTROUT DD DSN=&&TROUT,
//         SPACE=(6160,(10,10)),
//         DISP=(NEW,PASS),UNIT=SYSDA
//SYSIN    DD DSN=*.SYSTROUT,
//         VOL=REF=*.SYSTROUT,
//         DISP=(OLD,PASS)
//SYSUT1   DD UNIT=SYSDA,
//         SPACE=(TRK,(10,10))
//SYSUT2   DD UNIT=SYSDA,
//         SPACE=(TRK,(10,10))
//SYSUT3   DD UNIT=SYSDA,
//         SPACE=(TRK,(10,10))
//SYSTRDB  DD UNIT=SYSDA,
//         SPACE=(TRK,(10,10))
//SYSLIN   DD DSN=&&OBJECT,
//         SPACE=(3200,(10,10)),
//         DISP=(MOD,PASS),UNIT=SYSDA,
//         DCB=(RECFM=FB,LRECL=80)
//SYSLIB   DD DSN=SASC.MACLIBC,
//      DISP=SHR C++ AND C STANDARD HEADERS
//SYSDBLIB DD DSN=&&DBGLIB,
//         SPACE=(4080,(20,20,1)),
//         DISP=(,PASS), UNIT=SYSDA,
//         DCB=(RECFM=U,BLKSIZE=4080)
//SYSTMP01 DD UNIT=SYSDA,
//         SPACE=(TRK,25) VS1 ONLY
//SYSTMP02 DD UNIT=SYSDA,
//         SPACE=(TRK,25) VS1 ONLY
//LKED     EXEC PGM=COOLB,PARM='LIST,MAP',
//         REGION=1536K,COND=(8,LT,X)
//STEPLIB  DD DSN=SASC.LINKLIB,
//         DISP=SHR C RUNTIME LIBRARY
//         DD DSN=SASC.LOAD,
//         DISP=SHR COMPILER LIBRARY
//SYSPRINT DD SYSOUT=A,
//    DCB=(RECFM=FBA,LRECL=121,BLKSIZE=1210)
//SYSTERM  DD SYSOUT=A
//SYSIN    DD DSN=*.X.SYSLIN,
//         VOL=REF=*.X.SYSLIN,
//         DISP=(OLD,PASS)
//SYSLIN   DD UNIT=SYSDA,DSN=&&LKEDIN,
//         SPACE=(3200,(20,20)),
//    DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SYSLIB   DD DDNAME=AR#&ALLRES
//*        ARESOBJ OR STDOBJ OR SPEOBJ
//         DD DSN=SASC.&ENV.OBJ,
//    DISP=SHR STDOBJ OR SPEOBJ OR GOSOBJ
//         DD DSN=&SYSLIB,DISP=SHR
//*        COMMON RESIDENT LIBRARY
//         DD DSN=&CALLLIB,DISP=SHR
//SYSDBLIB DD DSN=*.X.SYSDBLIB,DISP=OLD,
//         VOL=REF=*.X.SYSDBLIB
//SYSARLIB DD DSN=&CXXLIB,DISP=SHR
//SYSUT1   DD DSN=&&SYSUT1,UNIT=SYSDA,
//         DCB=BLKSIZE=1024,
//         SPACE=(1024,(200,50))
//SYSLMOD  DD DSN=&&LOADMOD(MAIN),
//         DISP=(,PASS),UNIT=SYSDA,
//         SPACE=(1024,(50,20,1))
//AR#NO    DD DSN=SASC.&ENV.OBJ,DISP=SHR
//AR#YES   DD DSN=SASC.ARESOBJ,DISP=SHR
```

JCL for the LCXXL cataloged procedure

This procedure links an already translated and compiled program.

```
//LCXXL PROC ENV=STD,ALLRES=NO,
//      CALLLIB='SASC.BASEOBJ',
//      SYSLIB='SASC.BASEOBJ'
//      CXXLIB='SASC.LIBCXX.A'
//*******************************************
//* NAME: LCXXL
//* SUPPORT: C COMPILER DIVISION
//* PRODUCT: SAS/C C++ DEVELOPMENT SYSTEM
//* PROCEDURE: COOL, LINK EDIT
//* DOCUMENTATION: SAS/C C++ DEVELOPMENT
//*                SYSTEM USER'S GUIDE
//* FROM: SAS INSTITUTE INC., SAS CAMPUS DR.,
//*       CARY, NC 27513
//*******************************************
//*
//*******************************************
//* ENV=STD: MODULE RUNS IN THE NORMAL C
//*          ENVIRONMENT
//* ENV=CICS: MODULE RUNS IN A CICS
//*           C ENVIRONMENT
//* ENV=GOS: MODULE RUNS IN THE GENERALIZED
//*          OPERATING SYSTEM ENVIRONMENT
//* ENV=SPE: MODULE USES THE SYSTEMS
//*          PROGRAMMING ENVIRONMENT
//********************************************
//LKED     EXEC PGM=COOLB,PARM='LIST,MAP',
//         REGION=1536K
//STEPLIB  DD DSN=SASC.LINKLIB,
//         DISP=SHR C RUNTIME LIBRARY
//         DD DSN=SASC.LOAD,
//         DISP=SHR COMPILER LIBRARY
//SYSPRINT DD SYSOUT=A,
//    DCB=(RECFM=FBA,LRECL=121,BLKSIZE=1210)
//SYSTERM  DD SYSOUT=A
//SYSLIN   DD UNIT=SYSDA,DSN=&&LKEDIN,
//         SPACE=(3200,(20,20)),
//    DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SYSLIB   DD DDNAME=AR#&ALLRES
//*        ARESOBJ OR STDOBJ OR SPEOBJ
//         DD DSN=SASC.&ENV.OBJ,
//    DISP=SHR STDOBJ OR SPEOBJ OR GOSOBJ
//         DD DSN=&SYSLIB,DISP=SHR
//*        COMMON RESIDENT LIBRARY
//         DD DSN=&CALLLIB,DISP=SHR
//SYSDBLIB DD DSN=&&DBGLIB,
//         SPACE=(4080,(20,20,1)),
//         DISP=(,PASS),UNIT=SYSDA,
//         DCB=(RECFM=U,BLKSIZE=4080)
//SYSARLIB DD DSN=&CXXLIB,DISP=SHR
//SYSUT1   DD DSN=&&SYSUT1,UNIT=SYSDA,
//         DCB=BLKSIZE=1024,
//         SPACE=(1024,(200,50))
//SYSLMOD  DD DSN=&&LOADMOD(MAIN),
//         DISP=(,PASS),UNIT=SYSDA,
//         SPACE=(1024,(50,20,1))
//AR#NO    DD DSN=SASC.&ENV.OBJ,DISP=SHR
//AR#YES   DD DSN=SASC.ARESOBJ,DISP=SHR
```

JCL for the LCXXCLG cataloged procedure

This procedure translates, compiles, links, and executes your program.

```
//LCXXCLG PROC ENV=STD,ALLRES=NO,
//        CALLLIB='SASC.BASEOBJ',
//        MACLIB='SASC.MACLIBC',
//        SYSLIB='SASC.BASEOBJ',
//        CXXLIB='SASC.LIBCXX.A'
//*******************************************
//* NAME: LCXXCLG
//* SUPPORT: C COMPILER DIVISION
//* PRODUCT: SAS/C C++ DEVELOPMENT SYSTEM
//* PROCEDURE: TRANSLATE, COMPILE, COOL,
//*            LINK EDIT, GO
//* DOCUMENTATION: SAS/C C++ DEVELOPMENT
//*                SYSTEM USER'S GUIDE
//* FROM: SAS INSTITUTE INC., SAS CAMPUS DR.,
//*       CARY, NC 27513
//*******************************************
//*
//*******************************************
//* ENV=STD: MODULE RUNS IN THE NORMAL C
//*          ENVIRONMENT
//* ENV=CICS: MODULE RUNS IN A CICS
//*           C ENVIRONMENT
//* ENV=GOS: MODULE RUNS IN THE GENERALIZED
//*          OPERATING SYSTEM ENVIRONMENT
//* ENV=SPE: MODULE USES THE SYSTEMS
//*          PROGRAMMING ENVIRONMENT
//*******************************************
//X        EXEC PGM=LC370CX
//STEPLIB  DD DSN=SASC.LOAD,
//         DISP=SHR TRANSLATOR LIBRARY
//         DD DSN=SASC.LINKLIB,
//         DISP=SHR C RUNTIME LIBRARY
//SYSTERM  DD SYSOUT=A
//SYSPRINT DD SYSOUT=A
//SYSTROUT DD DSN=&&TROUT,
//          SPACE=(6160,(10,10)),
//          DISP=(NEW,PASS),UNIT=SYSDA
//SYSIN     DD DSN=*.SYSTROUT,
//          VOL=REF=*.SYSTROUT,
//          DISP=(OLD,PASS)
//SYSUT1    DD UNIT=SYSDA,
//          SPACE=(TRK,(10,10))
//SYSUT2    DD UNIT=SYSDA,
//          SPACE=(TRK,(10,10))
//SYSUT3    DD UNIT=SYSDA,
//          SPACE=(TRK,(10,10))
//SYSLIN    DD DSN=&&OBJECT,
//          SPACE=(3200,(10,10)),
//          DISP=(MOD,PASS),UNIT=SYSDA,
//          DCB=(RECFM=FB,LRECL=80)
//SYSLIB    DD DSN=&MACLIB,
//      DISP=SHR C++ AND C STANDARD HEADERS
//SYSDBLIB DD DSN=&&DBGLIB,
//          SPACE=(4080,(20,20,1)),
//          DISP=(,PASS),UNIT=SYSDA,
//          DCB=(RECFM=U,BLKSIZE=4080)
//SYSTMP01 DD UNIT=SYSDA,
//          SPACE=(TRK,25) VS1 ONLY
//SYSTMP02 DD UNIT=SYSDA,
//          SPACE=(TRK,25) VS1 ONLY
//LKED      EXEC PGM=COOLB,PARM='LIST,MAP',
//          REGION=1536K,COND=(8,LT,X)
//STEPLIB   DD DSN=SASC.LINKLIB,
//          DISP=SHR C RUNTIME LIBRARY
//          DD DSN=SASC.LOAD,
//          DISP=SHR COMPILER LIBRARY
//SYSPRINT DD SYSOUT=A,
//    DCB=(RECFM=FBA,LRECL=121,BLKSIZE=1210)
//SYSTERM   DD SYSOUT=A
//SYSIN     DD DSN=*.X.SYSLIN,
//    VOL=REF=*.X.SYSLIN,DISP=(OLD,PASS)
//SYSLIN    DD UNIT=SYSDA,DSN=&&LKEDIN,
//          SPACE=(3200,(20,20)),
//    DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SYSLIB    DD DDNAME=AR#&ALLRES
//*         ARESOBJ OR STDOBJ OR SPEOBJ
//          DD DSN=SASC.&ENV.OBJ,
//    DISP=SHR STDOBJ OR SPEOBJ OR GOSOBJ
//          DD DSN=&SYSLIB,DISP=SHR
//*         COMMON RESIDENT LIBRARY
//          DD DSN=&CALLLIB,DISP=SHR
//SYSDBLIB DD DSN=*.X.SYSDBLIB,DISP=OLD,
//          VOL=REF=*.X.SYSDBLIB
//SYSARLIB DD DSN=&CXXLIB,DISP=SHR
//SYSUT1    DD DSN=&&SYSUT1,UNIT=SYSDA,
//          DCB=BLKSIZE=1024,
//          SPACE=(1024,(200,50))
//SYSLMOD   DD DSN=&&LOADMOD(MAIN),
//          DISP=(,PASS),UNIT=SYSDA,
//          SPACE=(1024,(50,20,1))
//AR#NO     DD DSN=SASC.&ENV.OBJ,DISP=SHR
//AR#YES    DD DSN=SASC.ARESOBJ,DISP=SHR
//GO        EXEC PGM=*.LKED.SYSLMOD,
//          COND=((8,LT,X),(4,LT,LKED))
//STEPLIB   DD DSN=SASC.LINKLIB,
//          DISP=SHR C TRANSIENT LIBRARY
//SYSPRINT DD SYSOUT=A
//SYSTERM   DD SYSOUT=A
//DBGTERM   DD SYSOUT=A
//DBGLOG    DD SYSOUT=A
//DBGLIB    DD DSN=*.X.SYSDBLIB,
//          DISP=(OLD,PASS),
//          VOL=REF=*.X.SYSDBLIB
//SYSTMPDB DD UNIT=SYSDA,
//          SPACE=(TRK,25) VS1 ONLY
```

JCL for the LCXXLG cataloged procedure

This procedure links and executes an already translated and compiled program.

```
//LCXXLG PROC ENV=STD,ALLRES=NO,
//        CALLLIB='SASC.BASEOBJ',
//        SYSLIB='SASC.BASEOBJ',
//        CXXLIB='SASC.LIBCXX.A'
//*******************************************
//* NAME: LCXXLG
//* SUPPORT: C COMPILER DIVISION
//* PRODUCT: SAS/C C++ DEVELOPMENT
//* PROCEDURE: COOL, LINK EDIT, GO
//* DOCUMENTATION: SAS/C C++ DEVELOPMENT
//*                SYSTEM USER'S GUIDE
//* FROM: SAS INSTITUTE INC., SAS CAMPUS DR.,
//*       CARY, NC 27513
//*******************************************
//*
//*******************************************
//* ENV=STD: MODULE RUNS IN THE NORMAL C
//*          ENVIRONMENT
//* ENV=CICS: MODULE RUNS IN A CICS
//*           C ENVIRONMENT
//* ENV=GOS: MODULE RUNS IN THE GENERALIZED
//*          OPERATING SYSTEM ENVIRONMENT
//* ENV=SPE: MODULE USES THE SYSTEMS
//*          PROGRAMMING ENVIRONMENT
//*******************************************
//LKED      EXEC PGM=COOLB,PARM='LIST,MAP',
//          REGION=1536K
//STEPLIB   DD DSN=SASC.LINKLIB,
//          DISP=SHR C RUNTIME LIBRARY
//          DD DSN=SASC.LOAD,
//          DISP=SHR COMPILER LIBRARY
//SYSPRINT DD SYSOUT=A,
//     DCB=(RECFM=FBA,LRECL=121,BLKSIZE=1210)
//SYSTERM   DD SYSOUT=A
//SYSLIN    DD UNIT=SYSDA,DSN=&&LKEDIN,
//          SPACE=(3200,(20,20)),
//      DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SYSLIB    DD DDNAME=AR#&ALLRES
//*         ARESOBJ OR STDOBJ OR SPEOBJ
//          DD DSN=SASC.&ENV.OBJ,
//      DISP=SHR STDOBJ OR SPEOBJ OR GOSOBJ
//          DD DSN=&SYSLIB,DISP=SHR
//*         COMMON RESIDENT LIBRARY
//          DD DSN=&CALLLIB,DISP=SHR
//SYSDBLIB DD DSN=&&DBGLIB,
//          SPACE=(4080,(20,20,1)),
//          DISP=(,PASS),UNIT=SYSDA,
//          DCB=(RECFM=U,BLKSIZE=4080)
//SYSARLIB DD DSN=&CXXLIB,DISP=SHR
//SYSUT1   DD DSN=&&SYSUT1,UNIT=SYSDA,
//          DCB=BLKSIZE=1024,
//          SPACE=(1024,(200,50))
//SYSLMOD  DD DSN=&&LOADMOD(MAIN),
//          DISP=(,PASS),UNIT=SYSDA,
//          SPACE=(1024,(50,20,1))
//AR#NO    DD DSN=SASC.&ENV.OBJ,DISP=SHR
//AR#YES   DD DSN=SASC.ARESOBJ,DISP=SHR
//GO       EXEC PGM=*.LKED.SYSLMOD,
//          COND=(4,LT,LKED)
//STEPLIB  DD DSN=SASC.LINKLIB,
//          DISP=SHR C TRANSIENT LIBRARY
//SYSPRINT DD SYSOUT=A
//SYSTERM  DD SYSOUT=A
//DBGTERM  DD SYSOUT=A
//DBGLOG   DD SYSOUT=A
//SYSTMPDB DD UNIT=SYSDA,
//          SPACE=(TRK,25) VS1 ONLY
```

Translating C++ Programs under OpenEdition

The compiler drivers **sascc370** and **sasCC370** control the translation, compilation, and link-editing of C++ programs under OpenEdition. The following syntax is used to translate, compile, and/or link a C++ program from the OpenEdition shell:

sascc370 [*options*] *filename1* [*filename2* ...]
sasCC370 [*options*] *filename1* [*filename2* ...]

The *options* argument is a list of SAS/C Compiler options, COOL prelinker options, and/or MVS linkage editor options (for details, see the *SAS/C Compiler and Library User's Guide*). You can also specify the C++ translator options as described in "Option Descriptions" on page 38.[*] It also lists the SAS/C Compiler options that are not valid for C++ compilations. The *filename* arguments may contain any combination of C++ source files, C source files, object modules, and **ar370** archives. **sascc370** invokes the C++ translator if any input file has a **.C** or **.cxx** extension, or if you specify the **-cxx** compiler option. **sasCC370** invokes the C++ translator if any input file has a **.c**, **.C**, or **.cxx** extension.

If you do not suppress the prelinking step with the **-c** compiler option, both **sascc370** and **sasCC370** invoke the COOL prelinker followed by the linkage editor to link the object files. The C++ library is automatically added to the prelinking step if any C++ translations have occurred.

The input source files may reside either in the OpenEdition hierarchical file system (HFS) or in a standard MVS partitioned data set. The following example illustrates the use of an HFS file:

```
sasCC370 ./proj/sort.cxx
```

If this program was located in an MVS PDS member named YOURLOG.PROJ4.CXX(SORT), the compiler could be invoked from the OpenEdition shell as follows:

```
sasCC370 '//dsn:yourlog.proj4.cxx(sort)'
```

In either case, the compiled and linked output module is stored in the file **a.out** in your current directory. To specify another file, use the **-o** option. For example, the following command stores the output module in the file **./proj5/sort**:

```
sasCC370 -o ./proj5/sort ./proj5/sort.cxx
```

To invoke the **sascc370** or **sasCC370** command, you must include the directory where SAS/C was installed in your PATH environment variable. Typically, your site will define PATH appropriately for you when you start up the shell. If your site does not define PATH for you, contact your SAS Software Representative for C compiler products to obtain the correct directory name and add it to your PATH.

For additional information about compiling programs under the OpenEdition shell, see the *SAS/C Compiler and Library User's Guide*.

COOL Control Statements

COOL accepts four control statements: INCLUDE, INSERT, ARLIBRARY, and GATHER. All COOL control statements must have a blank in the first column.

INCLUDE Statement

The INCLUDE control statement is used to specify one or more additional files to be used as input to COOL. The INCLUDE statement has two formats. The first is

INCLUDE *filename* <, . . .>

Under TSO and MVS batch, *filename* is a DDname that has been allocated to a sequential data set or PDS member. Under CMS, *filename* is the filename of a CMS file. The filetype must be TEXT. The file can be on any accessed minidisk or in a SFS directory accessed as a minidisk, or it can be in an SFS directory that has been named in the _INCLUDE environment variable. (See the *SAS/C Compiler and Library User's Guide* for more information about the _INCLUDE environment variable.)

The second format of the INCLUDE statement is

INCLUDE *libname* (*member*<,*member*>)<, . . .>

Under CMS, *libname* is the name of a TEXT library. The filetype must be TXTLIB. The library can be on any accessed disk. *member* is the name of a member in the TEXT library. Under TSO and MVS batch, *libname* is a DDname that has been allocated to a PDS, and *member*

[*] C++ translator options are processed by the translator itself, not by the compiler.

is the name of a member in that PDS. The two formats can be combined in the same statement. Here are several examples of the INCLUDE statement:

INCLUDE MYPROJ
: Under CMS, the file MYPROJ TEXT is used as input. Under TSO and MVS batch, this statement causes the sequential data set allocated to the MYPROJ DDname to be used as input.

INCLUDE PROJLIB(MAINPROG)
: Under CMS, member MAINPROG of PROJLIB TXTLIB is used. Under TSO and MVS batch, this statement causes member MAINPROG in the PDS allocated to the PROJLIB DDname to be used as input.

INCLUDE PROJLIB(MAINPROG,SUBRTN)
: Under CMS, both member SUBRTN and member MAINPROG of PROJLIB TXTLIB are used. Under TSO and MVS batch, this statement causes member SUBRTN in the PDS allocated to the PROJLIB DDname to be used as input, in addition to member MAINPROG.

INCLUDE PROJLIB(MAINPROG),SUBRTN
: Under CMS, member MAINPROG of PROJLIB TXTLIB and the file SUBRTN TEXT are used as input. Under TSO and MVS batch, this statement causes member MAINPROG in the PDS allocated to the PROJLIB DDname to be used as input. In addition, the sequential data set allocated to the SUBRTN DDname is used.

An included object file can also contain an INCLUDE statement. The specified modules are also included, but any data in the object file after the INCLUDE statement is ignored.

INSERT Statement

The INSERT control statement is used to specify one or more external symbols that are to be resolved, if necessary, via COOL's autocall mechanism. The format of the INSERT statement is

INSERT *symbol-1<,symbol-2 . . .>*

If a symbol specified in an INSERT statement is not resolved after all primary input has been processed, COOL attempts to resolve it by using automatic library call.

ARLIBRARY Statement

The ARLIBRARY statement specifies the names of one or more AR370 archives to be used to resolve external references by COOL. The format of the ARLIBRARY statement is as follows:

ARLIBRARY *name-1<,name-2 . . .>*

Under TSO and MVS batch, *name* is the DDname that has been allocated to the data set. Under CMS, *name* is the filename of the archive file. COOL uses A as the filetype.

For more information on the AR370 utility, refer to the *SAS/C Compiler and Library User's Guide*.

GATHER Statement

The last control statement accepted by COOL is the GATHER control statement. You do not need to use any GATHER statements to link your C++ program; however, a GATHER statement is generated automatically for you by COOL when C++ programs are linked. This special statement causes COOL to create tables of functions that are used to initialize and terminate static objects in your program. COOL prints the names of the functions in these tables in its listing. Usually this information can be ignored, although it may be useful for debugging.

The format of the GATHER statement is

GATHER *prefix<,prefix2 . . .>*

where *prefix* is a one- to six-character symbol.

COOL Options

Table 2.2 on page 30 lists the options available for the COOL utility and the systems to which these options apply. The majority of these options are documented in detail in the *SAS/C Compiler and Library User's Guide*. This table is primarily for reference only. A select few of the COOL options (the ones most applicable to the C++ environment) are documented following the table.

Table 2.2 COOL Options

Option	TSO	CMS	MVS Batch
allowre-cool	X	X	X
allresi-dent	X	X	
arlib	X		
auto		X	
cics	X	X	
cicsvse	X	X	
cxx	X	X	
dbglib	X	X	X
dupsname	X	X	X
enexit	X	X	X
enexitdata	X	X	X
entry	X		
enxref	X	X	X
extname	X	X	X
files			X

Option	TSO	CMS	MVS Batch
genmod		X	
global		X	
gmap	X	X	X
gos	X	X	
ignorere-cool	X	X	X
inceof	X	X	X
lib	X		
libe		X	
lineno	X	X	X
list	X	X	X
lked		X	
lkedname	X		X
load	X		
loadlib	X		
nocool	X		X
output		X	
pagesize	X	X	X
prem	X	X	X
print	X	X	
prmap	X	X	X
rtconst	X	X	X
smpjclin	X	X	X
smponly	X	X	X
smpxivec	X	X	X
spe	X	X	
start		X	
term	X	X	X
upper	X	X	X
verbose	X	X	X
warn	X	X	X
xfnmkeep	X	X	X
xsymkeep	X	X	X

The following list describes the COOL options that are most applicable to the C++ environment.

allowrecool
specifies that the output object deck can be reprocessed by COOL. Therefore, the deck is not marked as already processed by COOL.

The default **noallowrecool** specifies that the output object cannot be reprocessed by COOL. A later attempt to reprocess the deck with COOL will produce an error.
The short form of this option is **-rc**.

Note: COOL does not modify the object deck to enable reprocessing. It is the user's responsibility to determine if a particular object is eligible for reprocessing. △

See *SAS/C Software: Changes and Enhancements, Release 6.50* for more information on this option.

cxx
specifies that the LCXX CLIST or EXEC should add the C++ object library to the autocall list. This option is valid under TSO and CMS.
The short form of this option is **-cxx**.

dbglib
specifies a debugger file qualifier that provides for customization of the destination of the debugger file. For each platform, **dbglib** specifies something different:

On MVS:
A SAS/C file specification that denotes a PDS. The filename is constructed using whatever is supplied, followed by (***sname***)

On CMS:
If the option specified starts with a ' **/**' , then it is assumed that this is either a ' **//sf:**' file specification or an SFS path. In this case, the specification is prepended to the filename. For example:

```
dbglib(//sf:/ted/)
```

will generate the name

```
//sf:/ted/sname.DB
```

If the option specified does not start with a ' **/**', then it is considered to be a filemode, and will be appended to the filename. For example:

```
dbglib(d2)
```

will generate the name

```
sname.db.d2
```

The option has different defaults on the various platforms:

On MVS:
dbglib(ddn:sysdblib)

On CMS:
dbglib(A)
For the various platforms, this default causes the filename to take different forms:

On MVS:
ddn:sysdblib(*sname*)

On CMS:
***sname*.DB.A**
The short form of this option is **-db**.

enxref
controls the production of the cross-references that are produced by default when COOL produces object files that contain extended names. The default value is **noenxref**.

If **enxref** is specified, COOL produces three cross-references that are generated in a table that follows all other COOL output. These three cross-references are SNAME, CID, and LINKID. SNAME is in alphabetical order by the SNAME that uniquely identifies an object file. CID displays the extended names in alphabetical order by C identifier. LINKID displays the extended names in alphabetical order by a link ID that COOL assigns. When REFERENCES is specified, referenced symbols as well as defined symbols are included in the cross-reference listing. **noenxref** suppresses the production of all extended names cross-references.

Under TSO, the **enxref** option takes the following form:

```
enxref('cross-ref,cross-ref,cross-ref')
```

where *cross-ref* is SNAME, CID, or LINKID, or its negation. For example, the following specification suppresses the SNAME cross-reference and enables the CID cross-reference:

```
enxref('nosname,cid')
```

Under CMS, the **enxref** option takes the following form:

```
enxref <cross-ref> <cross-ref> <cross-ref>
```

where *cross-ref* is SNAME, CID, or LINKID, or its negation. For example, the following specification suppresses the SNAME cross-reference and enables the CID cross-reference:

```
enxref nosname cid
```

Under CMS, if you specify **enxref** with no arguments, all three listings are produced.

Under MVS batch, the **enxref** option takes the following form:

```
enxref(cross-ref,cross-ref,cross-ref)
```

where *cross-ref* is SNAME, CID, or LINKID, or its negation. For example, the following specification suppresses the SNAME cross-reference and enables the CID cross-reference:

```
enxref(nosname,cid)
```

The short forms of this option are:

-xxx	for **enxref(*cid*)**
-xxe	for **enxref(*linkid*)**
-xxs	for **enxref(*sname*)**
-xxy	for **enxref(*references*)**

genmod
specifies that the COOL EXEC is to create a MODULE file named ***filename***, using the **genmod** options that are specified. The **genmod** option takes the following form:

```
genmod <filename <options>>
```

The **genmod** option must follow any use of any other option on the command line. The **genmod** option causes the COOL EXEC to issue the following CMS commands after COOL has created the COOL370 TEXT file:

```
LOAD COOL370 (NOAUTO NOLIBE CLEAR
GENMOD filename
```

where *filename* is either the filename specified following the **genmod** keyword or the first name specified in the COOL command. If no filenames are specified in the command, the COOL EXEC issues an error message. The **genmod** option is valid only under CMS.

ignorerecool
specifies that if any marks are detected indicating that COOL has already processed an input object deck, then the marks are to be ignored. If the **ignorerecool** option is specified along with the **verbose** option, then a diagnostic message is issued and processing continues.

The default **noignorerecool** specifies that any marks indicating that COOL has already processed an input object deck should result in an error message and process termination.

The short form of this option is **-ri**.

See *SAS/C Software: Changes and Enhancements, Release 6.50* for more information on this option.

lib
specifies the data set name of an autocall object library containing functions that are to be linked automatically into the program if referenced. The **lib** keyword has the following form:

```
lib (dsname)
```

(Note that load module libraries cannot be used.) If the library belongs to another user, the fully qualified name of the data set must be given, and the name must be preceded and followed by three apostrophes. No final qualifier is assumed for a **lib** data set. This option is valid only under TSO.

load
names the data set into which the linkage editor stores the output load module. The **load** keyword has the following form:

```
load (dsname)
```

This keyword should specify a PDS member. If the data set belongs to another user, the fully qualified name of the data set must be given and the name must be preceded and followed by three apostrophes. If the data set name is not specified within three apostrophes, it is assumed to be a data set name with a final qualifier of LOAD. Additional information about the **load** option is available in the *SAS/C Compiler and Library User's Guide*. This option is valid only under TSO.

print
controls where COOL and linkage editor output listings should be printed. The **print** keyword has two forms. The first form is as follows:

```
print(*)
```

This form of the **print** keyword indicates that the COOL and linkage editor output listings should be printed at the terminal.

The other form of the **print** keyword is as follows:

print (*dsname*)

This form of the **print** keyword specifies that the COOL and linkage editor listings should be stored in the named data set. This data set must be sequential; a PDS member is not allowed. If the data set belongs to another user, the fully qualified name of the data set must be given, and the name must be preceded and followed by three apostrophes. If the data set name is not specified within three apostrophes, it is assumed to be a data set name with a final qualifier of LINKLIST. **noprint** specifies that no linkage editor or COOL listing is to be produced. If you use the **noprint** option with CLK370, COOL, and linkage editor output, except for diagnostic messages, is suppressed. If neither **print** nor **noprint** is used, the default is **noprint**. This option is valid only under TSO.

The short form of this option is **-h**.

term
specifies that COOL error messages be written to the standard error output file (the terminal or the SYSTERM DD statement in MVS batch) as well as to COOL's standard output. **noterm** suppresses COOL's error output.

Under TSO, **term** is the default if the **print** option is not used, or if the **print** option specifies a data set. The default is **noterm** if **print(*)** is specified.

Under CMS and MVS batch, the default is **noterm**.

The short form of this option is **-t**.

upper
produces all output messages in uppercase. This option is valid in TSO, CMS, and MVS batch.

The short form of this option is **-u**.

warn
specifies that warning messages (which are associated with RC=4) are to be issued. This is the default. **nowarn** suppresses warning messages. This option is valid in TSO, CMS, and MVS batch.

The short form of this option is **-w**.

CHAPTER 3 Translator Options

Introduction

The translator accepts a number of options that enable you to alter the behavior of the translator. This chapter explains what options are available and how to specify them in each environment (CMS, TSO, MVS batch, and the OpenEdition shell).

Remember that when you invoke the translator, first your C++ code is translated to C and is then compiled by the SAS/C Compiler. You can specify options when you invoke the translator. The syntax of specifying options in each of the supported environments is covered in Chapter 2, "Using the SAS/C C++ Development System under TSO, CMS, MVS Batch, and OpenEdition," on page 13. The translator inspects each of the options you specify and decides if it needs to act on that option. The only options that are acted upon by the translator are those with a T in the Affects column in Table 3.1 on page 36. Options without a T in this column are passed to the compiler at the compilation step. These options affect the C code output by the compiler. Some options are both acted upon by the translator and then also passed on to the compiler. These options have both a T and a C in the Affects column in Table 3.1 on page 36.

Which Options to Use

The translator accepts any compiler option described in the *SAS/C Compiler and Library User's Guide*, except for the **cxx** option, which is implied when you invoke the translator. Table 3.1 on page 36 lists those options that are of special interest to C++ users, including the translator-specific options **savec** and **tronly** and those options whose behavior is slightly different when used with the translator (such as **pponly**).

Note: If you invoke the translator using a SAS/C Compiler option that is not documented in this book, you do not receive an error. However, some compiler options, such as listing options, apply only to the C code generated by the translator, not to your C++ code. Other compiler options are inappropriate for C++ and should not be used when translating and compiling a C++ program. For example, because C++ requires the presence of prototypes for all functions, the **reqproto** option is inappropriate. As another example, the **japan** option has no effect when used with C++. △

Listing options

Release 6.50 of the translator generates a listing of the C++ source code. By default, source listing, options listings, and so on are produced. That is, the defaults for C++ are **print**, **options**, **source**, and **noxref**. You can use all of the listing options accepted by the SAS/C Compiler with the translator.

You can combine listing options such as **ilist** and **maclist** with options that prevent compilation such as **tronly** and **pponly**. Because the listing options are completely documented in the *SAS/C Compiler and Library User's Guide*, that information is not repeated here.

Options that Affect Warning and Error Messages

The translator performs the majority of the diagnostic work. Therefore, any options such as **enforce** and **suppress** that control how messages are generated apply to messages generated by the translator, not the compiler.

Option Summary

Table 3.1 on page 36 summarizes all options accepted by the translator that are appropriate to the C++ environment.

In Table 3.1 on page 36, the option name is in the first column. An asterisk after the option name indicates that the option may function differently depending on the environment in which it is used. The second column lists the default for each option. The third column indicates how the option is specified from the OpenEdition shell. The next column lists the environment(s) for which an option is implemented. If the option is valid in TSO, CMS, and MVS batch, this column indicates "all". The

Affects Processes column names the process that the option affects:

C	affects compilation.
G	affects global optimization.
L	affects listings.
M	affects message generation.
T	affects C++ translation.
X	affects cross referencing.

Detailed information about each option follows the table. Note that under MVS batch, the OpenEdition shell, and CMS, if you specify contradictory options, the option specified last is used. Under TSO, the options are concatenated and treated as a single invalid option.

Table 3.1 Translator Options

Option Name	Default	OpenEdition	Environment	Affects Processes
ALias	**NOALias**	**-Kalias**	all	G
ARlib*	see description		TSO	C
AT	**NOAT**	**-Kat**	all	T,C
AUtoinst	**NOAUtoinst**	**-Kautoinst**	all	T,C
BItfield	**4**	**-Kbitfield=*n***	all	T,C
BYtealign	**NOBYtealign**	**-Kbytealign**	all	T,C
COMPlexity*	**0**	**-Kcomplexity=*n***	all	G
DBGmacro	**NODBGmacro**	**-Kdbgmacro**	all	C
DBGObj	**NODBGObj**	**-Kdbgobj**	all	C
DEBug	**NODEBug**	**-Kdebug[=*filename*]**	all	T,C
DEFine*	see description	**-D[*sym=val*]**	all	T
DEPth	**3**	**-Kdepth=*n***	all	G
DIgraph	see description	**-Kdigraph[*n*]**	all	T,C
DLines	**NODLines**		all	T
DOllars	**NODOllars**	**-Kdollars**	all	T,C
ENforce*	see description	**-w~*n***	all	M
FIles*	see description		MVS batch	T,C
FReg	**2**	**-Kfreg=*n***	all	G
GReg	**6**	**-Kgreg=*n***	all	G
HList	**NOHList**	**-Khlist**	all	L
HXref	**NOHXref**	**-Khxref**	all	X
IList	**NOIList**	**-Kilist**	all	L
HMulti	**HMulti**	**-Khmulti**	all	T
IMulti	**IMulti**	**-Kimulti**	all	T
INDep	**NOINDep**	**-Kindep**	all	C
INLIne	see description	**-Kinline**	all	G
INLOcal	**NOINLOcal**	**-Kinlocal**	all	G

Option Name	Default	OpenEdition	Environment	Affects Processes
IXref	**NOIXref**	**-Kixref**	all	X
LIB*	**NOLIB**		TSO	T,C
LINeno	**LINeno**	**-Klineno**	all	C
LOop	**LOop**	**-Kloop**	all	G
MAClist	**NOMAClist**	**-Kmaclist**	all	L
MEMber*	see description		TSO	C
MENtion	see description	**-w+*n***	all	M
OBject*	see description	**-o *filename***	TSO, CMS	C
OLDFORSCOPE	**OLDFORSCOPE**	**-Koldforscope**	all	T
OPTIMize	see description	**-Koptimize**	all	G
Overload	**NOOverload**	**-Koverload**	all	T
OVERStrike	**NOOVERStrike**	**-Koverstrike**	all	L, X
PAgesize	**PAgesize(55)**	**-Kpagesize=*nn***	all	L, X
PFlocal	**NOPFlocal**	**-Kpflocal**	all	T,C
POsix	see description	**-Kposix**	TSO, MVS batch	C
PPOnly*	**NOPPOnly**	**-P**	all	T
PRInt	see description	**-Klisting[=*filename*]**	all	L, X
RDEpth	**0**	**-Krdepth=*n***	all	G
REDef	**NOREDef**	**-Kredef**	all	T
REFdef	**NOREFdef**	**-Krefdef**	all	C
RENT	**NORENT**	**-Krent**	all	T,C
RENTExt	**NORENTExt**	**-Krentext**	all	T,C
RTTI	**NORTTI**	**-Krtti**	all	T
SAvec*	**NOSAvec**		all	T
SIze	**NOSIze**		all	G
SName*	see description	**-Ksname=*sname***	all	T,C
SOurce	**SOurce**	**-Ksource**	all	L
STRICt	**NOSTRICt**	**-Kstrict**	all	M
STRINgdup	**STRINgdup**	**-Kstringdup**	all	C
SUppress*	see description	**-w*n***	all	M
TIme	**NOTIme**		all	G
TMplfunc	**NOTMplfunc**	**-Ktmplfunc**	all	T
TRAns	**TRAns**	**-Ktrans**	all	L, X
TRIgraphs	**NOTRIgraphs**	**-Ktrigraphs**	all	T
TRonly*	see description		all	T
UNdef*	**NOUNdef**	**-Kundef**	all	T
UPper	**NOUPper**	**-Kupper**	all	L, X
Warn	**Warn**	**-Kwarn**	all	M

Option Name	Default	OpenEdition	Environment	Affects Processes
Xref	**NOXref**	**-Kxref**	all	X
ZAPMin*	**ZAPMin(24)**	**-Kzapmin=*n***	all	C
ZAPSpace*	**1**	**-Kzapspace=*n***	all	C

Option Descriptions

The following list gives detailed descriptions of the options listed in Table 3.1 on page 36.

alias (**-Kalias** under OpenEdition)
specifies that the global optimizer should assume worst-case aliasing. See the

```
optimize
```

option in the *SAS/C Compiler and Library User's Guide* for details on this option. This option can only be used in conjunction with the **optimize** option.

arlib
identifies an AR370 archive in which the generated object code is to be stored. **arlib** is valid under TSO only and cannot be specified together with the **object** option.
The **arlib** option is specified as follows:

```
arlib(dsname)
```

where *dsname* specifies the name of an AR370 archive. If the archive belongs to another user, you must specify the fully qualified name of the data set, and the name must be preceded and followed by three apostrophes, as in the following example:

```
arlib('''master.object.a''')
```

The extra apostrophes are required by the CLIST language. If the data set name is not enclosed within three apostrophes, it is assumed to be a data set with a final qualifier of A.
You can use the **member** option to specify the archive member name in which the object code is to be stored. If **arlib** is specified and **member** is not specified, the default member name is the partitioned data set (PDS) member name of the source file. If the source file is not a PDS member, you must supply a member name if you use the **arlib** option.

at (**-Kat** under OpenEdition)
allows the use of the call-by-reference operator @.

autoinst (**-Kautoinst** under OpenEdition)
controls automatic implicit instantiation on template functions and static data members of template classes. The compiler organizes the output object module so that COOL can arrange for only one copy of each template item to be included in the final program. To correctly perform the instantiation, the **autoinst** option must be enabled on a compilation unit that contains both a use of the item and its corresponding template definition.

Note: Automatic instantiation requires that the translation of the C++ code to C code must occur as part of the same process that generates the object module. Therefore, while the **savec** and the **tronly** options can be used with the **autoinst** option, the resulting C code cannot be compiled with the C compiler to produce an equivalent object module at a later time. △

bitfield (**-Kbitfield=*n*** under OpenEdition)
enables you to specify the allocation unit to be used for plain **int** bitfields. (C++, unlike C, supports bitfields that are not integers.)
The following allocation units are valid:

1	indicates the allocation unit is a **char**.
2	indicates the allocation unit is a **short**.
4	indicates the allocation unit is a **long**.

The default allocation unit is a **long** (4).
Under TSO and MVS batch, the **bitfield** option is specified as follows:

```
bitfield(value)
```

For example, the following option specification indicates the allocation unit for **int** bitfields is a short:

```
bitfield(2)
```

Under CMS, the **bitfield** option is specified as follows:

```
bitfield value
```

For example, the following option specification indicates the allocation unit for **int** bitfields is a **long**:

```
bitfield 4
```

This option cannot be negated.

bytealign (**-Kbytealign** under OpenEdition)
aligns all data on byte boundaries. Most data items, including all those in structures, are generated with only character alignment. Because formal parameters are aligned according to normal IBM 370 conventions even when you specify the **bytealign** option, you can call functions compiled with byte alignment from functions that are not compiled with byte alignment and vice versa.
You can attach the **__noalignmem** keyword to structure definitions to force the structure to be byte-aligned. Use the **__alignmem** keyword to force structures to be normally aligned even in modules compiled with the **bytealign** option.
If functions compiled with and without byte alignment are to share the same structures, you must en-

sure that such structures have exactly the same layout. The layout is not exactly the same if any structure element does not fall on its usual boundary; for example, an **int** member's offset from the start of the structure is not divisible by 4. You can force such alignment by adding unreferenced elements of appropriate length between elements as necessary. If a shared structure does contain elements with unusual alignment, you must compile all functions that reference the structure using byte alignment.

complexity (**-Kcomplexity=*n*** under OpenEdition)
specifies the maximum "complexity" the function can have and remain eligible for default inlining. This option applies to functions that have not been defined using the **inline** keyword from C++ or the **__inline** keyword from SAS/C and is used only in conjunction with the **optimize** option. See the **optimize** option in the *SAS/C Compiler and Library User's Guide* for more details.

dbgmacro (**-Kdbgmacro** under OpenEdition)
specifies that definitions of C macro names should be saved in the debugger file. Note that this substantially increases the size of the file.

dbgobj (**-Kdbgobj** under OpenEdition)
causes the compiler to place the debugging information in the object, instead of a separate debugger file. Debugging of automatically instantiated compiled objects will not work when the debugging information is not placed in the object.

If automatic instantiation is specified with the **autoinst** option, **dbgobj** is enabled automatically.

By default, the **dbgobj** option is off. The short form for the option is **-xc**.

debug (**-Kdebug[=*filename*]** under OpenEdition)
allows the use of the SAS/C Debugger. Note that the **debug** option causes the compiler to suppress all optimizations as well as store and fetch variables to or from memory more often.

define (**-D[*sym*=*val*]** under OpenEdition)
defines a symbol to an (optional) value.

Under MVS batch and CMS, you can use the **define** option more than once, to define any number of symbols. If the same symbol is defined twice, only the last value applies.

Under TSO, the **define** option can be used only once. If you specify this option more than once, only the last specification is used. Also note that TSO uppercases the text of the **define** option before it is passed to the translator.

Under TSO, the specification is the following:

```
define(symbol)
define(symbol=value)
```

Here is an example:

```
define(USERDATA)
define(TSO=1)
```

Under CMS, the specification is the following:

```
define symbol
define symbol=value
```

Here are some examples:

```
define USERDATA
define CMS=1
```

Under MVS batch, the **define** option is specified as follows:

```
define(symbol)
```

Here are some examples:

```
define(USERDATA)
define(MYSYM=ABC)
```

depth (**-Kdepth=*n*** under OpenEdition)
specifies the maximum depth of function calls to be inlined. This option is used only with the **optimize** option. See the **optimize** option in the *SAS/C Compiler and Library User's Guide* for more information.

Specify **depth** as follows, where *n* is between 0 and 6, inclusive (the default is 3):

System	Syntax
MVS batch	**depth (*n*)**
TSO	**depth(*n*)**
CMS	**depth *n***

digraph (**-Kdigraph[*n*]** under OpenEdition)
enables the translation of the International Standard Organization (ISO) digraphs and the SAS/C digraph extensions.

Specify **digraph** as follows:

System	Syntax
MVS Batch	**digraph(*n*)**
TSO	**digraph(*n*)**
CMS	**digraph *n***

where ***n*** is between 0 and 3, inclusive. Specify ***n*** as follows:

0	Turn off all digraph support.
1	Turn on New ISO digraph support.
2	Turn on SAS/C Bracket digraph support — '(|' or '|)'
3	Turn on all SAS/C digraphs. This alone does not activate the new ISO digraphs.

The default options are

```
digraph(1) and digraph(2)
```

See "Special characters" on page 5 and "Alternate forms for operators and tokens" on page 8 for more information.

dlines
suppresses emission of **#line** directives in the preprocessed C++ source code. This option has an effect only when the **pponly** option is in effect.

dollars (**-Kdollars** under OpenEdition)
allows the use of the **$** character in identifiers, except as the first character.*

enforce (**-w~*n*** under OpenEdition)
treats one or more translator warning messages as error messages. Each warning message is identified by an associated message number. Messages whose numbers have been specified are treated as errors, and the translator return code is set to 12 instead of 4.

Under TSO, specify the **enforce** option as follows:

```
enforce (n)
```

where *n* is the message number you want to enforce. If more than one warning message is to be enforced, specify each number in a comma-delimited list, enclosed by quotes, as follows:

```
enforce('n1,n2,...')
```

Under CMS, use the following:

```
enforce n
enforce n1 n2 ...
```

Under MVS batch, use the following:

```
enforce(n)
enforce(n1,n2,...)
```

Any number of warning messages can be specified. If both **suppress** and **enforce** specify the same warning message number, the warning is enforced.

files
replaces SYS in translator DDnames with the other prefix. This option is valid for MVS batch only. The only DDname in which SYS cannot be replaced is SYSTERM. The prefix can contain from one to three characters. For example, the following specification causes the SYS prefix to be replaced by CXX:

```
files(cxx)
```

In this case, some of the DDname replacements are as follows:

Original	Replacement
SYSTROUT	CXXTROUT

freg (**-Kfreg=*n*** under OpenEdition)
specifies the maximum number of floating-point registers that the optimizer can assign to register variables in a function. This option is used only with the *optimize* option. See the **optimize** option in the *SAS/C Compiler and Library User's Guide* for additional details.

Specify **freg** as follows, where *n* is between 0 and 2, inclusive (the default is 2):

System	Syntax
TSO	**freg(*n*)**
CMS	**freg *n***

greg (**-Kgreg=*n*** under OpenEdition)
specifies the maximum number of registers that the optimizer can assign to register variables in a function. This option is used only with the **optimize** option. See the **optimize** option in the *SAS/C Compiler and Library Guide* for more details.

Specify **greg** as follows, where *n* is between 0 and 6, inclusive (the default is 6):

System	Syntax
TSO	**greg(*n*)**
CMS	**greg *n***

hlist (**-Khlist** under OpenEdition)
includes standard header files in the formatted source listing. These files are included using the following syntax:

```
#include <name.h>
```

or

```
#include <name>
```

hmulti (**-Khmulti** under OpenEdition)
allows reinclusion of a header file specified within angle brackets. **hmulti** is the default. If **nohmulti** is used, then the translator does not reinclude a header file specified within angle brackets.

hxref (**-Khxref** under OpenEdition)
prints references in standard header files in the cross reference listing. See **hlist** for a description of header files.

ilist (**-Kilist** under OpenEdition)
includes user header files referenced by the **#include** statement in the formatted source listing. The **#include** filename appears in the right margin of each line taken from the **#include** file. See also **hlist**

imulti (**-Kimulti** under OpenEdition)
allows reinclusion of a header file specified within double quotes. **imulti** is the default. If **noimulti** is used, then the translator does not reinclude a header file specified within double quotes.

* If you use the all-resident library and the **resident.h** header file with your C++ program, you must specify the **dollars** option.

indep (**-Kindep** under OpenEdition)
generates code that can be called before the C framework is initialized or code that can be used for interlanguage communication. See the *SAS/C Compiler and Library User's Guide* for a detailed description of the use of this option.

inline (**-Kinline** under OpenEdition)
inlines small functions identified by **complexity** and those with the C++ **inline** keyword or the SAS/C **__inline** keyword. This option is used only with the **optimize** option. See the **optimize** option in the *SAS/C Compiler and Library User's Guide* for more details.

inlocal (**-Kinlocal** under OpenEdition)
inlines single-call **static** functions. This option is used only with the **optimize** option. See the **optimize** option in the *SAS/C Compiler and Library User's Guide* for more information.

ixref (**-Kixref** under OpenEdition)
lists references in user **#include** files.

lib
identifies a header file library and is valid under TSO only. The **lib** option is specified as follows:

```
lib(dsname)
```

This option indicates the name of a library that contains header files, that is, one containing members that are to be included using the **#include <member.h>** (or **<member>**) form of the **#include** statement. If the library belongs to another user, the fully qualified name of the data set must be used and the name must be preceded and followed by three apostrophes (because of CLIST language requirements). No final qualifier is assumed for a **lib** data set.

Using **nolib** indicates that no header file libraries are required other than the standard library provided with the translator. **nolib** is the default.

lineno (**-Klineno** under OpenEdition)
allows identification of source lines in run-time messages. (When **lineno** is specified, module size is increased because of the generation of line number and offset tables.)

loop (**-Kloop** under OpenEdition)
specifies that the global optimizer should perform loop optimizations. See the **optimize** option in the *SAS/C Compiler and Library User's Guide* for more details on this option. This option is used only with the **optimize** option.

maclist (**-Kmaclist** under OpenEdition)
prints macro expansions. Source code lines containing macros are printed before macro expansion.

member
is used to specify the member of an AR370 archive in which the object code is to be stored. **member** is valid under TSO only. The **member** option is specified as follows:

```
member(member-name)
```

where *member-name* is a valid MVS member name.

The **member** option is valid only if the **arlib** option is also specified. Otherwise, **member** is ignored.

If **arlib** is specified and **member** is not specified, the default member name is the PDS member name of the source file. If the source file is not a PDS member, you must supply a member name if you use the **arlib** option.

mention (**-w+*n*** under OpenEdition)
specifies that the translator warnings whose numbers are specified as *n1*, *n2*, and so on are not to be suppressed. (See the **suppress** option as well.)

Under TSO, specify the **mention** option as follows:

```
mention(n)
```

where *n* is the number of the message associated with the warning condition. If more than one warning condition is to be mentioned, specify the numbers in a comma-delimited list, enclosed by quotes, as follows:

```
mention('n1,n2,...')
```

Under CMS, use the following:

```
mention n mention n1 n2 ...
```

Under MVS batch, use the following:

```
mention(n)
mention(n1,n2...)
```

Any number of warning conditions can be specified.

object (**-o *filename*** under OpenEdition)
outputs object code.

Under TSO, this option is specified as follows:

```
object(dsname)
```

where *dsname* names the data set in which the compiler stores the object code. The data set name can be a PDS member. If the data set belongs to another user, the fully qualified name of the data set must be specified and the name must be preceded and followed by three apostrophes, as in the following example:

```
OBJECT('''YOURLOG.PROJ4.OBJ(PART1)''')
```

The extra apostrophes are required for the CLIST language. If the data set name is not specified within three apostrophes, it is assumed to be a data set with a final qualifier of OBJ.

Using **noobject** indicates that no object code is to be stored by the compilation.

When neither **object** nor **noobject** is specified under TSO, the default depends on how the source data set name is specified, as explained here:

- If the source data set name is specified in apostrophes, the default is **noobject**.
- Otherwise, the default is **object**. (The object data set name is determined by replacing the final CXX in the source data set name with OBJ.)

Under TSO, if both **noobject** and **omd** are specified, object code is generated but discarded after the OMD is run. See the *SAS/C Compiler and Library User's Guide* for a discussion of the **omd** option.

Under CMS, the default is **object**. By default, object code is generated in pass two of the compiler. If you specify **noobject**, pass two is suppressed and object code is not generated. Under CMS, if both **noobject** and **omd** are specified, neither pass two nor the OMD is run.

oldforscope (**-Koldforscope** under OpenEdition)
specifies that the scope of a variable defined in the initialization clause of a **for** statement will follow the old rules concerning scoping. The new ANSI scoping rules specify that the scope of a variable defined in the **for** loop initialization clause only includes the **for** statement and its associated loop body. Therefore, the code in the following example would not work under the new scoping rules:

```
for (int i=0; i < n; ++i)
    if ( f(i) )
    break;
if ( i < n ) // the 'i' declared in
             // the 'for' loop
   do_something(); // broke out of
                   //the loop...
```

For Release 6.50, **oldforscope** is the default for compatibility with previously written code.

optimize (**-Koptimize** under OpenEdition)
executes the global optimizer phase of the compiler. Optimizing is the default unless you use the **debug** option. See the **optimize** option in the *SAS/C Compiler and Library User's Guide* for details on this option.

overload (**-Koverload** under OpenEdition)
turns on the recognition of the **overload** C++ keyword. If this option is on, the translator recognizes the keyword syntax as documented (for example, in Stroustrup's *The C++ Programming Language, Second Edition*). The **overload** keyword is obsolete in modern C++. The **overload** keyword is treated as a reserved word only if the **overload** option is turned on; it is treated as an identifier otherwise.

overstrike (**-Koverstrike** under OpenEdition)
prints special characters in the listing file as overstrikes. This option is useful, for example, if you do not have a printer that can print the special characters left brace, right brace, left bracket, right bracket, and tilde. See the *SAS/C Compiler and Library User's Guide* for more information on special characters.

pagesize (**-Kpagesize=*nn*** under OpenEdition)
defines the number of lines per page of source and cross reference listings. **pagesize** is specified as follows:

System	Syntax
MVS batch	**pagesize (*nn*)**
TSO	**pagesize (*nn*)**
CMS	**pagesize *nn***

nn lines per page of listing are printed at the location determined by the **print** option. The default is 55 lines per page. (The default location is different for each operating system and is described in the discussion of **print**.)

pflocal (**-Kpflocal** under OpenEdition)
assumes that all functions are **__local** unless **__remote** is explicitly specified in the declaration. The default is **nopflocal**.

posix (**-Kposix** under OpenEdition)
informs the compiler that the program is POSIX-oriented and that compile-time and run-time defaults should be changed for maximum POSIX compatability. The default is **noposix** under TSO, CMS, and MVS batch and **---Kposix** under OpenEdition.

Specifically, the **posix** option has the following effects on compilation:

- ☐ The SAS/C feature test macro **_SASC_POSIX_SOURCE** is automatically defined.
- ☐ The translator option **refdef** is assumed if **norefdef** is not also specified.
- ☐ The special POSIX symbols **environ** and **tzname** are automatically treated as **__rent** unless declared as **__norent**.

Additionally, if any compilation in a program's main load module is compiled with the **posix** option, it will have the following effects on the execution of the program:

- ☐ The **fopen** function assumes at run-time that all filenames are HFS filenames unless prefixed by **"//"**.
- ☐ The **system** function assumes at run-time that the command string is a shell command unless prefixed by **" // "**.
- ☐ The **tmpfile** and **tmpnam** functions refer to HFS files in the **/tmp** directory.

Note that you should not use the **posix** translator option when compiling functions that can be used by both POSIX and other applications that are not POSIX.

Under OpenEdition and UNIX, use **-Kposix**.

For details about developing POSIX applications, see the *SAS/C Compiler and Library User's Guide.*

pponly (**-P** under OpenEdition)
creates a file containing preprocessed source code for this translation. Preprocessed source code has all macros and **#include** files expanded. If the **pponly** option is used, all syntax checking (except in preprocessor directives) is suppressed, no listing file is produced, and no object code is generated. The preprocessor used by the **pponly** option is the C++ preprocessor, not the C

preprocessor. These two preprocessors are identical, except that the C++ preprocessor accepts C++ style comments as well as C style comments.

Under TSO, use the following:

```
pponly(dsname)
```

where *dsname* indicates the name of a data set in which the preprocessed source file is to be stored. If the library belongs to another user, the fully qualified name of the data set must be used and the name must be preceded and followed by three apostrophes because of the CLIST language requirements. No final qualifier is assumed for a **pponly** data set.

Under CMS, use **pponly**. The output file is written to a file with the same filename as the source file and a filetype of PP.

Under MVS batch, use **pponly**. The output file is written to the data set allocated to the DDname SYSTROUT. Because the default SYSTROUT data set is temporary, you should alter the SYSTROUT DD statement to refer to a permanent file.

print (**-Klisting[=*filename*]** under OpenEdition)
produces a listing file.

Under MVS batch, the **print** option produces a listing file and sends it to SYSPRINT. The listing file also includes error messages. If **noprint** is used, the listing file is suppressed. Under MVS batch, the default is **print**.

In TSO, the **print** option is used with both the LCXX CLIST and the OMD370 CLIST to specify where the listing file is to be stored.

If you specify the following, the listing file is printed at the terminal:

```
print (*)
```

If you use **print (*)**, you do not need to use the **term** compiler option. If you do, error messages are sent to the terminal twice.

The following stores the listing file in the named data set:

```
print (dsname)
```

This data set must be sequential; a partitioned data set member is not allowed. If the data set belongs to another user, the fully qualified name of the data set must be specified, and the name must be preceded and followed by three single quotes because of the CLIST language requirements. If the data set name is not specified within three quotes, it is assumed to be a data set with a final qualifier of LIST.

The following form specifies that no listing file is to be produced:

```
noprint
```

If you use **noprint**, the compiler ignores all other listing options, such as **pagesize** and **ilist**. The **xref** option also is ignored.

If the source data set name is enclosed in single quotes, the default is **noprint**. Otherwise, the default is **print**. The listing data set name is determined by replacing the final CXX in the source data set name with LIST and ignoring any member name specification.

You cannot specify **noprint** when you use the OMD370 CLIST.

If you do not specify **print** when you use the OMD370 CLIST, the default is **print (*)** if the object data set name is enclosed by single quotes. Otherwise, the listing data set name is determined by replacing the final OBJ qualifier in the source data set name with LIST, and any member name specification is ignored.

Under CMS, **print** spools the listing file to the printer. **noprint** suppresses the listing file. **noprint** is an alternative to the **print** option. You can also give the **print** option to the OMD370 EXEC.

Under OpenEdition, by default, no listing file is generated unless you specify the **-Klisting** option. You can supply the name of the listing file by specifying **-Klisting=*filename***. If **-Klisting** is specified without a filename, the listing is stored in an HFS file with a **.lst** extension. See the **object** option for a description of this process.

rdepth (**-Krdepth=*n*** under OpenEdition)
defines the maximum level of recursion to be inlined (the default is 0). This option is used only with the **optimize** option. See the **optimize** option in the *SAS/C Compiler and Library User's Guide* for more details. **rdepth** is specified as follows:

System	Syntax
TSO	**rdepth(*n*)**
CMS	**rdepth *n***

redef (**-Kredef** under OpenEdition)
allows redefinition and stacking of **#define** names.

refdef (**-Krefdef** under OpenEdition)
causes the compiler to generate code that forces the use of the strict ref/def model for reentrant external variables. The default is **norefdef**, which specifies that the compiler use the common model. Note that this option has meaning only when used in conjunction with the **rent** or **rentext** options. When **norent** is used, the compiler always uses the strict ref/def model; this cannot be overridden by the user.

rent (**-Krent** under OpenEdition)
allows reentrant modification of **static** and external data. If you use the **tronly** option (translate only) and use the **rent** option as well, be sure to specify the **rent** option at compile time.

rentext (**-Krentext** under OpenEdition)
allows reentrant modification of external data. If you use the **tronly** option (translate only) and use the **rentext** option as well, be sure to specify the **rentext** option at compile time.

rtti (**-Krtti** under OpenEdition)
enables the generation of information required for RTTI on class objects that have virtual functions. By default,

this option is not enabled because it increases the number and the size of the tables used to implement virtual function calls.

If your program uses the **dynamic_cast** or **typeid()** operators, the **rtti** option must be specified for each compilation unit to assure the class objects have the information required for dynamic type identification.

savec

creates a file containing the C source code emitted by the translator for this compilation. Translated source code has all macros and **#include** files expanded. Unless you use the **savec** option, the translator output is stored in a temporary data set and is discarded after compilation.

Under TSO, use the following:

```
savec(dsname)
```

where *dsname* indicates the name of a data set in which the translated source file is to be stored. If the library belongs to another user, the fully qualified name of the data set must be used, and the name must be preceded and followed by three apostrophes because of the CLIST language requirements. No final qualifier is assumed for a **savec** data set.

Under CMS, use

```
savec fileid
```

where *fileid* is any valid CMS filename. Under CMS, **savec** must be the last option on the command line; the remainder of the command line is interpreted as the fileid of the output file.

Under MVS batch, you cannot use the **savec** option. The output file is, by default, written to the data set allocated to the DDname SYSTROUT, which is normally a temporary data set. To override the default SYSTROUT DD statement, use your own DD statement, specifying a permanent file.

Note: If you compile the C source saved by the **savec** option with the SAS/C Compiler, you must compile with the **cxx** compiler option, which informs the compiler that it is compiling C code resulting from C++ translation. △

size

specifies that the global optimizer should favor optimizations that reduce the size of the generated code. This option is used only with the **optimize** option. See the **optimize** option in the *SAS/C Compiler and Library User's Guide* for details on this option.

sname (**-Ksname=*sname*** under OpenEdition)

defines the **sname** used by the translator and compiler. The name cannot be longer than seven characters. If *name* is longer than seven characters, it is truncated. The name cannot contain any national characters and can contain a dollar sign ($) only if you also specify the **dollars** option.

Each source file in a multisource file program should be translated using a unique value for the **sname** option. CLINK detects duplicate **snames** and terminates the link process. Usually, the default **sname** is sufficient to ensure uniqueness. For more information on **snames**, see the *SAS/C Compiler and Library User's Guide*.

Under TSO and MVS batch, the specification is as follows:

```
sname(name)
```

where *name* defines the **sname** and is unique to this source file. If you do not use the **sname** option, the **sname** defaults to the member name of the source file if it is a PDS member, or to NOSNAME otherwise.

Under CMS, the specification is as follows:

```
sname name
```

where *name* defines the **sname** and is unique to this source file. If you do not use the **sname** option, the **sname** defaults to the filename of the source file.

source (**-Ksource** under OpenEdition)

outputs a formatted source listing of the program to the listing file. (The default location of the listing file is different for each operating system and is described in the discussion of **print**.)

nosource suppresses only the source listing; the cross reference listing is still printed if requested with the **xref** option.

The **source** option has no effect on the OMD listing if an OMD listing is requested. Whether source code is merged into the OMD listing is controlled by the **merge** compiler option.

strict (**-Kstrict** under OpenEdition)

enables an extra set of warning messages for questionable or nonportable code. For more information on error messages, see the *SAS/C Software Diagnostic Messages, Release 6.50*manual.

stringdup (**-Kstringdup** under OpenEdition)

creates a single copy of identical string constants.

suppress (**-w*n*** under OpenEdition)

ignores one or more translator warning messages. Each warning message is identified by an associated message number. Messages whose numbers have been specified are suppressed. No message is generated, and the translator return code is unchanged.

Under TSO, specify the **suppress** option as follows:

```
suppress(n)
```

where *n* is the number of the message associated with the warning condition. If more than one warning message is to be suppressed, specify the numbers in a comma-delimited list, enclosed by quotes, as follows:

```
suppress('n1,n2,...')
```

Under CMS, use the following:

```
suppress n
suppress n1 n2 ...
```

Under MVS batch, use the following:

```
suppress(n)
suppress(n1,n2 ...)
```

Any number of warning messages can be specified. If both **suppress** and **enforce** specify the same message number, the warning is enforced.

Note that **suppress** suppresses only translator messages, not messages generated by the compiler.

time
specifies that the global optimizer should favor optimizations that reduce the execution time of the generated code. This option is used only with the **optimize** option. See the **optimize** option in the*SAS/C Compiler and Library User's Guide* for details on this option.

tmplfunc (**-Ktmplfunc** under OpenEdition)
controls whether a nontemplate function declaration that has the same type as a template specialization refers to the same object as the specialization. When this option is specified, template specializations are considered distinct from nontemplates.

trans (**-Ktrans** under OpenEdition)
translates special characters to their listing file representations. Default representations for these characters are

Character	Listing File Representation
left brace	0x8b {
right brace	0x9b }
left bracket	0xad [
right bracket	0xbd]
not sign (exclusive or)	0x5f ¬
tilde	0xa1 ° (degree symbol)
backslash	0xbe ≠
vertical bar (exclusive or)	0x4f |
pound sign	0x7b #
exclamation point	0x5a !

If you specify **notrans**, all special characters are written out as they appear in the source data.

trigraphs (**-Ktrigraphs** under OpenEdition)
enables translation of ANSI Standard trigraphs. See the *SAS/C Compiler and Library User's Guide* for more information on trigraphs.

tronly
creates a file containing the C source code that is the result of translating the C++ source for this compilation. Processing is terminated at this stage and does not go on to compile the source. Translated source code has all macros and **#include** files expanded. The normal syntax and semantic analysis of the C++ code is performed, and warning or error messages are emitted as appropriate.

Under TSO, use the following:

```
tronly(dsname)
```

where *dsname* indicates the name of a data set in which the translated source file is to be stored. If the library belongs to another user, the fully qualified name of the data set must be used, and the name must be preceded and followed by three apostrophes because of the CLIST language requirements. No final qualifier is assumed for a **tronly** data set.

Under TSO, the **tronly**, **pponly**, and **savec** options are mutually exclusive.

Under CMS, use **tronly**. The output file is written to a file with the same filename as the source file and a filetype of C.

Under MVS batch, use **tronly**. The output file is written to the data set allocated to the DDname SYSTROUT. Because the default SYSTROUT data set is temporary, you should alter the SYSTROUT DD statement to refer to a permanent file.

Note: If you compile the C source saved by the **tronly** option with the SAS/C Compiler, you must compile with the **cxx** compiler option, which informs the compiler it is compiling C code resulting from C++ translation. △

undef (**-Kundef** under OpenEdition)
undefines predefined macros. "Predefined constants" on page 7 describes the predefined macros for TSO, CMS, and MVS batch.

upper (**-Kupper** under OpenEdition)
outputs all lowercase characters as uppercase in the listing file. **upper** implies **overstrike**.

warn (**-Kwarn** under OpenEdition)
lists translation warning messages. **nowarn** suppresses warning messages.

xref (**-Kxref** under OpenEdition)
produces a cross reference listing.

zapmin (**-Kzapmin=*n*** under OpenEdition)
specifies the minimum size of the patch area, in bytes. Under TSO and MVS batch, use the following:

```
zapmin(n)
```

where *n* refers to the number of bytes in the patch area. The default is 24 bytes.

Under CMS, use the following:

```
zapmin n
```

where *n* refers to the number of bytes in the patch area. The default is 24 bytes.

For more information about the patch area, see the *SAS/C Compiler and Library User's Guide*. For more information about using the **zapmin** option, see the

zapmin option in the *SAS/C Compiler and Library User's Guide*.

zapspace (**-Kzapspace=*n*** under OpenEdition)
changes the size of the patch area generated by the compiler.

Under MVS batch or TSO, use the following:

```
zapspace(factor)
```

Under CMS, use the following:

```
zapspace factor
```

For more information about the patch area, see the *SAS/C Compiler and Library User's Guide*. For more information about using the **zapspace** option, see the **zapspace** option in the *SAS/C Compiler and Library User's Guide*.

CHAPTER 4

Standard Libraries

Introduction

Two libraries are provided with this release of the SAS/C C++ Development System: the complex library and the streams library. The complex library provides for manipulation of complex numbers, while the streams library provides for input, output, and string formatting. This chapter first describes the fundamentals of using header files with C++, after which the complex and streams libraries are described. A discussion of the most basic aspects of C++ I/O and considerations for performing I/O on IBM 370 machines is contained in "C++ I/O" on page 53. This chapter is primarily a reference chapter. Appendix 2, "Header Files, Classes, and Functions" provides cross-reference to the material contained in this chapter.

Note: By including the header files provided by SAS/C, you can use all the SAS/C library functions in your C++ program. The only exceptions are the **`setjmp`** and **`longjmp`** functions. Using **`longjmp`** could cause destructors not to be called for automatic objects, producing unpredictable results. △

Header Files

Most C++ programs need to access some of the classes and functions contained in the libraries provided with the SAS/C C++ Development System. These classes and functions are declared in header files. In addition, you can have header files of your own. Chapter 2, "Using the SAS/C C++ Development System under TSO, CMS, MVS Batch, and OpenEdition," on page 13 , describes how header files are stored and named under your operating system. Here are some other tips to keep in mind when you use header files in C++ programs:

- always use the following form for standard files (those provided with the translator or compiler):
  < *header-filename*.h>
  Use this form for your own header files:
  "*header-filename*.h"
- always use the fully qualified filename for your own files.

Header Files Supplied with the Standard Libraries

The following header files are supplied with the standard libraries:

complex.h	**iostream.h**	**strstream.h**
bsamstr.h	**new.h**	**typeinfo.h**
fstream.h	**stream.h**	
iomanip.h	**stdiostream.h**	

Except for **new.h** , **stream.h** , and **typeinfo.h** , the contents of these header files are described in "I/O Class Descriptions" on page 60 . The contents of **stream.h** are described in "stream.h header file" on page 59 .

new.h Header File

The contents of **new.h** are as follows:

void* operator new(size_t size)
allocates a block of dynamic memory of size bytes. If insufficient free dynamic memory is available and a new-handler is currently defined, the new-handler is called. When the new-handler returns, **operator new** tries again to allocate memory, and if that fails, it again calls the new-handler. This process repeats until either **operator new** can allocate some dynamic memory or the new-handler is removed.

void * operator new(size_t size, void* location);
does not allocate memory, but instead ignores the **size** argument and simply returns the **location** argument as its result. This version of **operator new** is used to construct objects at a specified location.

void operator delete(void* ptr)
frees the block of dynamic memory pointed to by **ptr** .

void (*set_new_handler (void(*handler)()))()
defines a user-defined function (a new-handler) to handle out-of-memory conditions detected by **operator new** . The new-handler is called if **operator new** cannot allocate any dynamic memory. The new-handler function takes no arguments and returns **void** . The new-handler should free some memory before returning to its caller, or an endless loop could result. If it cannot free any memory, it should terminate the program via **exit** or **abort** . Another alternative is to remove itself as a new-handler by calling **set_new_handler** with NULL or a pointer to some other handler. **set_new_handler** returns the previous new-handler or NULL if there was not a previous new-handler.

typeinfo.h Header File

The class, functions, and types declared in the **typeinfo.h** header file are provided to identify and compare types, and to control run time error handling for Run Time Type Identification, or RTTI. The class **type_info** is declared in this header file. A description of the functions, operators, and types found in **class type_info** is included in this section. Release 6.50 of the SAS/C C++ Development System does not support exceptions so a mechanism corresponding to default handling of an exception is invoked for these errors. Descriptions of the run time error handling type **terminate_handler** and functions **set_terminate,** and **terminate** are also included in this section.

See Appendix 4 for more information on RTTI.

class type_info

Provides information on object types

SYNOPSIS

```
#include <typeinfo.h>

class type_info
{
public:
   virtual ~type_info();

   int operator==(const type_info& rhs) const;
   int operator!=(const type_info& rhs) const;

   int before(const type_info& rhs) const;

   const char* name() const;
};
```

DESCRIPTION

This class is declared in **<typeinfo.h>** . The RTTI operator **typeid()** returns a reference to a **const** object of this type. Such objects are created by the C++ implementation to provide an identifier for types. The class allows two **type_info** objects to be compared for type equivalence, or compared for order in an arbitrary collating sequence of types. It also allows a printable description of the represented type to be retrieved.

CONSTRUCTORS
This class has no public constructors.

DESTRUCTORS
The destructor is virtual, so the class is polymorphic itself.

NONVIRTUAL MEMBER FUNCTIONS

`int operator==(const type_info& rhs) const;`
returns 1 if the two **`type_info`** objects represent the same type, and returns 0 otherwise.

`int operator!=(const type_info& rhs) const;`
returns 0 if the two **`type_info`** objects represent the same type, and returns 1 otherwise.

`int before(const type_info& rhs) const;`
returns 1 if the type represented by this object precedes the type represented by **`rhs`** in an arbitrary collating sequence of types, and returns 0 otherwise. This order is arbitrary and may change in different program loads.

`const char* name() const;`
returns a pointer to a character string containing a human readable representation of the type represented by the **`type_info`** object. The function calls operator **`new`** to allocate working storage. The result is cached so it should not be deallocated by the user.

set_terminate, terminate_handler, and terminate

Provide runtime error handling for RTTI

SYNOPSIS

```
#include <typeinfo.h>

typedef void (*terminate_handler)();
terminate_handler set_terminate
    (terminate_handler new_handler);
void terminate();
```

DESCRIPTION
In this release, these functions and types are used to control run-time error handling for RTTI. (In ISO C++ they are part of a more general exception handling mechanism.):

`terminate_handler`
a type representing a handler function. A handler function is called when **`terminate()`** is called. The handler function must end the program without returning.

`old_handler = set_terminate(new_handler);`
Sets a user specified terminate handler which can be used to perform cleanup before terminating program execution. The new handler pointer must not be NULL. The old handler value is returned.

`terminate();`
calls the termination handler. This function is effectively called under various conditions when a C++ exception can not be handled. C++ exceptions are not supported in this release, but RTTI operations can detect erroneous conditions wh ich are normally handled by throwing an exception. In this release, the RTTI operations, in effect, call **`terminate()`** . The default handler calls **`abort()`**.

C++ Complex Library

The functions and operators pertaining to complex numbers are implemented by means of **`class complex`** and are contained in the C++ complex number mathematics library. The definition of **`class complex`** overloads the standard input, output, arithmetic, comparison, and assignment operators of C++, as well as the standard names of the exponential, logarithm, power, square root, and trigonometric functions (sine, cosine, hyperbolic sine, and hyperbolic cosine). Functions for converting between Cartesian and polar coordinates are also provided. This section describes the functions, classes, and operators found in **`class complex`** . Declarations for these functions and operators are contained in the header file **`complex.h`** . In the function descriptions, the form **`(a,b)`** is used to represent a complex number. This is equivalent to the mathematical expression **`a+bi`** .

Function Descriptions

The function descriptions for the complex library are organized as follows:

constructors and conversion operators
include constructors and conversion operators for complex variables.

cartesian and polar coordinate functions
include descriptions of **`abs()`** , **`arg()`** , **`conj()`** , **`imag()`**, **`norm()`** , **`polar()`** , and **`real()`** .

exponential, logarithmic, power, and square root functions
include descriptions of **`exp()`** , **`log()`** , **`pow()`** , and **`sqrt()`** .

trigonometric and hyperbolic functions
include descriptions of **`sin()`** , **`cos()`** , **`sinh()`** , and **`cosh()`** .

operators
include the operators available for the complex library (**+** , ***** , **==** , and so on).

complex I/O functions
provide complex I/O, that is, the insertion and extraction operators **<<** and **>>** .

Each set of function descriptions includes the following information:

□ a synopsis of the functions

- □ a brief description of the purpose of the functions
- □ details about diagnostics, if appropriate
- □ any appropriate cautionary information.

In the following descriptions, specific diagnostic information is given for a function or group of functions, where appropriate. However, more general diagnostic information is not included in this book. Also, some of the complex library functions call SAS/C math library functions. If you find you need more information on error handling for math functions, or if you need more information about functions called by the complex library functions, see the *SAS/C Library Reference*. If you want to use signal handlers to trap overflows, see the *SAS/C Library Reference, Volume 1* also.

complex()

Constructors and Conversion Operators

SYNOPSIS

```
#include <complex.h>
class complex
{
public:
   complex();
   complex(double real, double imag = 0.0);
};
```

DESCRIPTION

The following constructors are defined for **class complex** .

complex()
enables you to declare complex variables without initializing them. File-scope complex variables declared without an initializer have an initial value of **(0,0)** ; other uninitialized complex variables have an undefined initial value.

complex(double real, double imag = 0.0)
allows explicit initialization of complex variables. For example, the following two statements are valid:

```
complex c1(1.0, 2.0);
// The imaginary part is 0.0.
complex c2(1.0);
```

This constructor also allows for implicit conversion from arithmetic types to complex values. For example, the following two statements are valid:

```
complex c3 = 3.4; // c3 is (3.4, 0.0).
c3=10; // c3 is (10.0, 0.0).
```

Using this constructor, you can also create complex values within expressions. Here is an example:

```
// Uses complex::operator +
c2 = c3 + complex(1.2, 3.5);
```

abs, arg, conj, imag, norm, polar, real

Cartesian and Polar Functions

SYNOPSIS

```
#include <complex.h>
class complex
{
public:
   friend double abs(complex a);
   friend double arg(complex a);
   friend complex conj(complex a);
   friend double imag(complex a);
   friend double norm(complex a);
   friend complex polar(double r,
                        double t);
   friend double real(complex a);
};
```

DESCRIPTION

The following functions are defined for **class complex**, where **d** , **r** , and **t** are of type **double** and **a** and **z** are of type **complex** .

d = abs(a)
returns the absolute value (magnitude) of **a** .

d = arg(a)
returns the angle of **a** (measured in radians) in the half-open interval ($-\pi$ to π].

z = conj(a)
returns the conjugation of **a** . If a is **(x,y)** , then **conj(a)** is **(x,-y)** .

d = imag(a)
returns the imaginary part of **a** .

d = norm(a)
returns the square of the magnitude of **a** .

z = polar(r, t)
returns a **complex** . The arguments represent a pair of polar coordinates where **r** is the magnitude and **t** is the angle (measured in radians). **polar(r,t)** is defined by the formula **r*e** ^{i*t}.

d = real(a)
returns the real part of **a** .

exp, log, pow, sqrt

Exponential, Logarithmic, Power, and Square Root Functions

SYNOPSIS

```
#include <complex.h>
class complex
{
public:
   friend complex exp(complex a);
   friend complex log(complex a);
   friend complex pow(double a, complex b);
   friend complex pow(complex a, int b);
   friend complex pow(complex a, double b);
   friend complex pow(complex a,
                      complex b);
   friend complex sqrt(complex a);
};
```

DESCRIPTION

The following functions are overloaded by the C++ complex library, where **z** is of type **complex** and **a** and **b** are of the types indicated by the function prototypes in the SYNOPSIS.

z = exp(a)
returns **e** ^{a}.

z = log(a)
returns the natural logarithm of **a** .

z = pow(a, b)
returns **a** ^{b} .

z = sqrt(a)
returns the square root of **a** that is contained in the first or fourth quadrant of the complex plane.

DIAGNOSTICS

The **exp()** and **log()** functions have special diagnostic considerations.

exp() function

If overflow is caused by the real part of **a** being small or the imaginary part of **a** being large, then **exp(a)** returns **(0,0)** and **errno** is set to **ERANGE**

If the real part of **a** is large enough to cause overflow, **exp(a)** returns different values under the following conditions pertaining to the sine and cosine of the imaginary part of **a** .

In Table 4.1 on page 51 , the real portion of a complex number **a** depends on the **cos(imag(a))** and the imaginary part depends on the **sin(imag(a))**. HUGE corresponds to the largest representable **double**.

Table 4.1 Return Values for exp(a)

cos(imag(a))	**sin(imag(a))**	**exp(a) returns**
positive	positive	(HUGE, HUGE)
positive	negative or 0	(HUGE, -HUGE)
negative or 0	positive	(-HUGE, HUGE)
negative or 0	negative or 0	(-HUGE, -HUGE)

As you can see from this table, the translation is simple. If the cosine is positive, **exp(a)** returns HUGE for the real portion of **a** ; if the cosine is negative, **exp(a)** returns -HUGE for the real part. If the cosine is not positive, **exp(a)** returns -HUGE for the real part. The same rules hold true for the sine and the imaginary part of **a** . In all overflow cases, **errno** is set to **ERANGE**.

log() function

When **a** is **(0,0), log(a)** returns **(-HUGE,0)** and **errno** is set to **EDOM** .

sin, cos, sinh, cosh

Trigonometric and Hyperbolic Functions

SYNOPSIS

```
#include <complex.h>
class complex
{
public:
   friend complex sin(complex a);
   friend complex cos(complex a);
   friend complex sinh(complex a);
   friend complex cosh(complex a);
};
```

DESCRIPTION

The following functions are defined for **class complex**, where **a** and **z** are of type **complex** .

z = sin(a)
returns the sine of **a** .

z = cos(a)
returns the cosine of **a** .

z = sinh(a)
returns the hyperbolic sine of **a** .

z = cosh(a)
returns the hyperbolic cosine of **a** .

DIAGNOSTICS

sin(a) and **cos(a)** return **(0,0)** if the real part of **a** causes overflow. If the imaginary part of **a** is large enough to cause overflow, **sin(a)** and **cos(a)** return values as shown in Table 4.2 on page 51 and Table 4.3 on page 51 . HUGE corresponds to the largest representable **double** .

Table 4.2 Return Values for cos(a)

cos(real(a))	**sin(real(a))**	**cos(a)** returns
positive or 0	positive or 0	(HUGE, -HUGE)
positive or 0	negative	(HUGE, HUGE)
negative	positive or 0	(-HUGE, -HUGE)
negative	negative	(-HUGE, HUGE)

Table 4.3 Return Values for sin(a)

sin(real(a))	**cos(real(a))**	**sin(a)** returns
positive or 0	positive or 0	(HUGE, HUGE)
positive or 0	negative	(HUGE, -HUGE)
negative	positive or 0	(-HUGE, HUGE)
negative	negative	(-HUGE, -HUGE)

sinh(a) and **cosh(a)** return (0,0) if the imaginary part of **a** causes overflow. If the real part of **a** is large enough to cause overflow, **sinh(a)** and **cosh(a)** return values according to Table 4.4 on page 52 .

Table 4.4 Return Values for cosh(a) and sinh(a)

cos(imag(a))	**sin(imag(a))**	**cosh(a)** and **sinh(a)** both return
positive or 0	positive or 0	(HUGE, HUGE)
positive or 0	negative	(HUGE, -HUGE)
negative	positive or 0	(-HUGE, HUGE)
negative	negative	(-HUGE, -HUGE)

In all overflow cases, **errno** is set to **ERANGE** .

Complex Operators

Operators for the C++ Complex Library

SYNOPSIS

```
#include <complex.h>
class complex
{
public:
   friend complex operator +(complex a,
                             complex b);
   friend complex operator -(complex a);
   friend complex operator -(complex a,
                             complex b);
   friend complex operator *(complex a,
                             complex b);
   friend complex operator /(complex a,
                             complex b);
   friend complex operator /(complex a,
                             double d);
   friend int operator ==(complex a,
                          complex b);
   friend int operator !=(complex a,
                          complex b);
   void operator +=(complex a);
   void operator -=(complex a);
   void operator *=(complex a);
   void operator /=(complex a);
   void operator /=(double d);
};
```

DESCRIPTION

The usual arithmetic operators, comparison operators, and assignment operators are overloaded for complex numbers. The usual precedence relations among these operators are in effect. In the descriptions below, **a** and **b** are of type **complex** and **d** is of type **double** .

Arithmetic operators

The following are the arithmetic operators.

a + b
is the arithmetic sum of **a** and **b** .

-a
is the arithmetic negation of **a** .

a - b
is the arithmetic difference of **a** and **b** .

a * b
is the arithmetic product of **a** and **b** .

a/b and **a/d**
are the arithmetic quotient of **a** and **b** or **a** and **d** .

Comparison operators

The following are the comparison operators.

a == b
is nonzero if **a** is equal to **b** ; it is zero otherwise.

a != b
is nonzero if **a** is not equal to **b** ; it is zero otherwise.

Assignment operators

The following are the assignment operators.

a += b
assigns to **a** the arithmetic sum of itself and **b** .

a -= b
assigns to **a** the arithmetic difference of itself and **b** .

a *= b
assigns to a the arithmetic product of itself and **b**.

a /= b and **a /= d**
assign to **a** the arithmetic quotient of themselves and **b** or **d** .

CAUTION

The assignment operators do not yield a value that can be used in an expression. For example, the following construction is not valid:

```
complex a, b, c;
a = (b += c);
```

operator << and operator >>

Complex I/O Functions

SYNOPSIS

```
#include <complex.h>
class complex
{
public:
   ostream& operator <<(ostream& os,
                        complex c);
   istream& operator >>(istream& is,
                        complex& c);
};
```

DEFINITION

The following functions provide insertion and extraction capabilities for complex numbers.

ostream& operator << (ostream& os, complex c)

writes a complex number **c** to **os** . The output is formatted in the following manner:

```
(real-part,imag-part)
```

where *real-part* and *imag-part* are the real and imaginary parts of the complex number, respectively. Both *real-part* and *imag-part* are formatted as doubles. For more information, refer to the descrip tion of the **operator <<(ostream&, double)** in “class ostream” on page 75 . The formatting of *real-part* and *imag-part* is controlled by flags associated with the stream. See “enum format_state” on page 67 .

istream& operator >>(istream& is, complex& c)

reads a formatted complex number from **is** into **c** . The **istream** should contain the complex number to be read in one of these formats:

```
(real-part,imag-part)
(real-part)
```

where *real-part* and *imag-part* are the real and imaginary parts of the complex number, respectively. Both *real-part* and *imag-part* should be formatted as **double** s. For more information, refer to the description of the **operator >>(istream&, double&)** in “class istream” on page 72 . The formatting of *real-part* and *imag-part* is controlled by flags associated with the stream. See “enum format_state” on page 67 .

Remember the following when performing complex I/O:

- □ you must use the parentheses and comma for input
- □ you can use white space in your input but it is not significant.

If your input variable represents a real number such as 5e–2 or (502), the >> operator interprets it as a complex number with an imaginary part of 0.

DIAGNOSTICS

If the **istream** does not contain a properly formatted complex number, **operator >>** sets the **ios::failbit** bit in the stream’s I/O state.

EXAMPLES

Here is an example of using **operator <<** :

```
complex c(3.4,2.1);
cout << "This is a complex: " << c << endl;
```

This code writes the following string to **cout** :

```
This is a complex: (3.4,2.1)
```

Here is an example of using **operator >>** . Suppose **cin** contains **(1.2, 3.4)** . Then the following code reads the value **(1.2, 3.4)** and the value of **c** becomes **(1.2,3.4)** .

```
complex c;
cin >> c;
```

C++ I/O

The fundamental concepts of I/O in C++ are those of streams, insertion, and extraction. An input stream is a source of characters; that is, it is an object from which characters can be obtained (extracted). An output stream is a sink for characters; that is, it is an object to which characters can be directed (inserted). (It is also possible to have bidirectional streams, which can both produce and consume characters.) This section explains the basics of performing C++ I/O. For more details, refer to your C++ programming manual.

This section covers the following components of C++ I/O:

- □ insertion and extraction
- □ explanation of streams
- □ formatting I/O
- □ using manipulators in streams
- □ I/O status handling.

Insertion and Extraction

Insertion is the operation of sending characters to a stream, expressed by the overloaded insertion operator **<<**. Thus, the following statement sends the character **'x'** to the stream **cout** :

```
cout << 'x';
```

Extraction is the operation of taking characters from a stream, expressed by the overloaded extraction operator **>>** . The following expression (where **ch** has type **char**) obtains a single character from the stream **cin** and stores it in **ch** .

```
cin >> ch;
```

Although streams produce or consume characters, insertion and extraction can be used with other types of data. For instance, in the following statements, the characters **'1'** , **'2'** , and **'3'** are inserted into the stream **cout** , after which characters are extracted from **cin** , interpreted as an integer, and the result is assigned to **i** :

```
int i = 123;
cout << i;
cin >> i;
```

Insertion and extraction can be overloaded for user-defined types as well. Consider the following code:

```
class fraction
{
   int numer;
   unsigned denom;
   friend ostream& operator <<(ostream& os,
                               fraction& f)
   {
      return os << f.numer << '/' << f.denom;
```

```
    };
};
```

These statements define an insertion operator for a user-defined **fraction** class, which can be used as conveniently and easily as insertion of characters or **ints** .

Get and put pointers

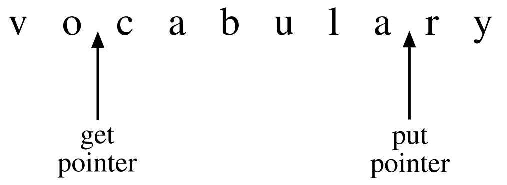
The definition of C++ stream I/O makes use of the concepts of the *get pointer* and the *put pointer*. The get pointer for a stream indicates the position in the stream from which characters are extracted. Similarly, the put pointer for a stream indicates the position in the stream where characters are inserted. Use of the insertion or extraction operator on a stream causes the appropriate pointer to move. Note that these are abstract pointers, referencing positions in the abstract sequence of characters associated with the stream, not C++ pointers addressing specific memory locations.

You can use member functions of the various stream classes to move the get or put pointer without performing an extraction or insertion. For example, the **fstream::seekoff()** member function moves the get and put pointers for an **fstream** .

The exact behavior of the get and put pointers for a stream depends on the type of stream. For example, for **fstream** objects, the get and put pointers are tied together. That is, any operation that moves one always moves the other. For **strstream** objects, the pointers are independent. That is, either pointer can be moved without affecting the other.

The get and put pointers reference positions between the characters of the stream, not the characters themselves. For example, consider the following sequence of characters as a stream, with the positions of the get and put pointers as marked in Figure 4.1 on page 54 :

Figure 4.1 Illustration of Get and Put Pointers



In this example, the next character extracted from the stream is ’ **c** ’, and the next character inserted into the stream replaces the ’ **r** ’.

Streams

This section covers the basics of using streams, including explaining which streams are provided by the streams library, how the different streams classes are related, which member functions are available for use with streams, and how to create your own streams.

Streams provided by the library

The streams library provides four different kinds of streams:

strstream
where characters are read and written to areas in memory.

fstream
where characters are read and written to external files.

stdiostream
where characters are read and written to external files using the C standard I/O library.

bsamstream
where a bsambuf object is used for performing formatted file I/O.

stdiostream objects should be used in programs that use the C standard I/O package as well as *C*++, to avoid interference between the two forms of I/O; on some implementations, **fstreams** provide better performance than **stdiostreams** when interaction with C I/O is not an issue.

Other types of streams can be defined by derivation from the base classes **iostream** and **streambuf** . See “Stream class hierarchy” on page 54 for more information on the relationships between these classes.

Every C++ program begins execution with four defined streams:

cin	is a standard source of input. It reads input from the same place that **stdin** would have.
cout	is a stream to which program output can be written. It writes output to the same place **stdout** would have.
cerr	is a stream to which program error output can be written. It writes output to the same place **stderr** would have.
clog	is another error stream that can be more highly buffered than **cerr** . It also writes output to the same place **stderr** would have.

To use these streams, you must include the header file **iostream.h** .

Additional streams can be created by the program as necessary. For more in- formation, see “Creating streams” on page 56 .

Stream class hierarchy

All the different stream classes are derived from two common base classes: **ios** and **streambuf** . **class ios** is a base class for the classes **istream** (an input stream), **ostream** (an output stream) and **iostream** (a bidirectional stream). You are more likely to use these classes as base classes than to use class **ios** directly. The **streambuf** class is a class that implements buffering for streams and controls the flushing of a full output buffer or the refilling of an empty input buffer.

Four sets of stream classes are provided in the standard streams library: **fstream** , **strstream** , **stdiostream**, and **bsamstream**. Corresponding to each of these stream classes is a buffer class: **`filebuf`** , **`strstreambuf`** , **`stdiobuf`** , and **`bsambuf`** , implementing a form of buffering appropriate to each stream. Note that the stream classes are not derived from the buffering classes; rather, a stream object has an associated buffering object of the appropriate kind (for instance, an **`fstream`** object has an associated **`filebuf`**), which can be accessed directly if necessary using the **`rdbuf()`** member function. Figure 4.2 on page 55 shows the inheritance relationships between the various classes.

Stream member functions

In addition to providing the insertion and extraction operations, the stream classes define a number of other member functions that can be more convenient than using insertion and extra ction directly. All these functions are discussed in some detail in the class descriptions later in this chapter. The following list briefly describes a few of the most useful member functions.

`get()` and **`getline()`**
: allows extraction of characters from a stream until a delimiter (by default ' **`\n`** ') is encountered, possibly with a limit to the number of characters to be extracted.

The **`get()`** and **`getline()`** functions behave similarly, except that **`getline()`** extracts the final delimiter and **`get()`** does not.

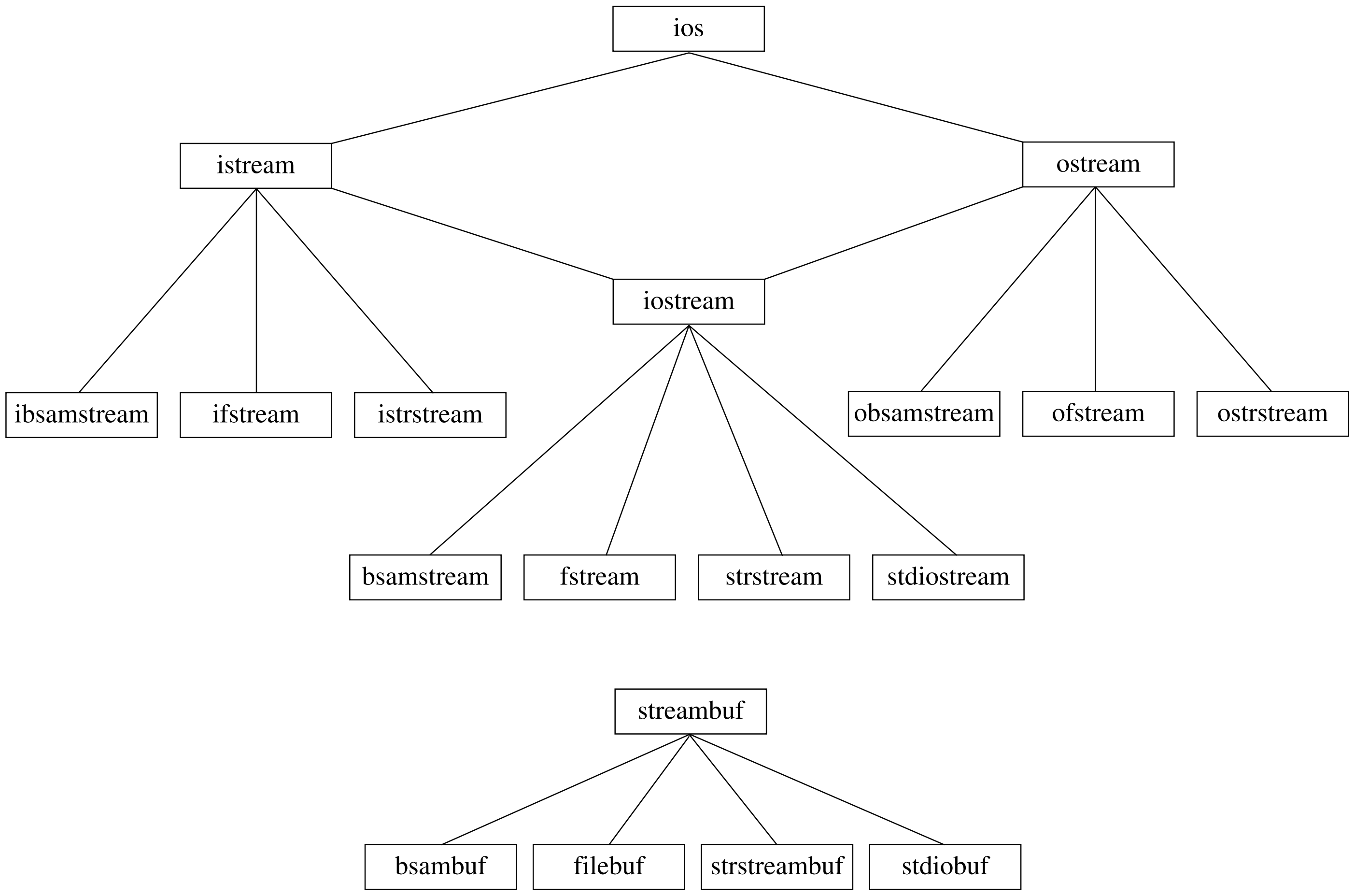
`read()`
: extracts a fixed number of characters from a stream. **`read()`** is intended for use with binary data, whereas **`get()`** and **`getline()`** are usually more appropriate with text data.

`putback()`
: allows a character extracted from a stream to be "pushed back," so that it will be extracted again the next time a character is required from the stream.

`write()`
: inserts a number of characters into a stream. The null character is treated as any other character and therefore this function is suitable for inserting binary data.

`flush()`
: immediately transmits any buffered characters. For a stream associated with a terminal file, this causes any buffered characters to be transmitted to the terminal. For a nonterminal file, calling **`flush()`** may not cause

Figure 4.2 Relationship between Stream Classes



any characters to be immediately written, depending on the characteristics of the file.

tie()
ties one stream to another stream, so that whenever the first file's buffer is full or needs to be refilled, the tied file's buffer is flushed. The **cin** stream is automatically tied to **cout** , which means that **cout** 's buffer is flushed before characters are extracted from **cin** . If, as is usually the case, **cin** and **cout** are both terminal files, this assures that you see any buffered output messages before having to enter a response. Similarly, the **cerr** stream is tied to **cout** , so that if an error message is generated to **cerr** , any buffered output characters are written first.

seekg() and **seekp()**
are used to reposition a stream for input and output respectively.

Note: Positioning of files is very different on 370 systems than on many other systems. See “370 I/O Considerations” on page 57 for some details. △

tellg() and **tellp()**
The member functions **tellg()** and **tellp()** are used to determine the read or write position for a stream. As with the seeking functions, the results of these functions are system-dependent. See “370 I/O Considerations” on page 57 for details.

Creating streams

Streams are normally created by declaring them or by use of the **new** operator. Creating an **fstream** or **stdiostream** entails opening the external file that is to be the source or sink of characters. Creating a **strstream** entails specifying the area of storage that will serve as the source or sink of characters. For **fstream** , a stream constructor can be used to create a stream associated with a particular file, similar to the way the **fopen** function is used in C. For instance, the following declaration creates an output **fstream** object, **dict** , associated with the CMS file named DICT DATA:

```
ofstream dict("cms:dict data");
```

Opening files

When you create an **fstream** (or a **stdiostream**), you must usually provide a filename and an open mode. The open mode specifies the way in which the file is to be accessed. For an **ifstream** , the default open mode is **ios::in** , specifying input only; for an **ofstream** , the default open mode is **ios::out** , specifying output only. See “370 I/O Considerations” on page 57 for information on additional arguments that can be supplied and for additional information on the form of filenames.

If you declare an **fstream** without specifying any arguments, a default constructor is called that creates an unopened **fstream** . An unopened stream can be opened by use of the member function **open()** , which accepts the same arguments as the constructor.

Defining a strstream

When you create a **strstream** , you must usually provide an area of memory and a length. Insertions to the stream store into the area of memory; extractions return successive characters from the area. When the array is full, no more characters can be inserted; when all characters have been extracted, the **ios::eof** flag is set for the stream. For an **istrstream** , the length argument to the constructor is optional; if you omit it, the end of the storage area is determined by scanning for an end-of-string delimiter (' **\0** ').

For a **strstream** that permits output, you can create a dynamic stream by using a constructor with no arguments. In this case, memory is allocated dynamically to hold inserted characters. When all characters have been inserted, you can use the member function **str()** to "freeze" the stream. This prevents further insertions into the stream and returns the address of the area where previously inserted characters have been stored.

Formatting

When a program inserts or extracts values other than single characters, such as integers or floating-point data, a number of different formatting options are available. For instance, some applications might want to have an inserted **unsigned int** transmitted in decimal, while for other applications hexadecimal might be more appropriate. A similar issue is whether white space should be skipped on input before storing or interpreting characters from a stream. The member function **setf()** is provided to allow program control of such options. For instance, the following expression sets the default for the stream **cout** to hexadecimal, so that integral values written to **cout** will ordinarily be transmitted in hexadecimal:

```
cout.setf(ios:hex, ios:basefield)
```

The streams library provides several similar functions:

width()
sets the number of characters to display.

fill()
defines the fill character when there are fewer characters to insert than the width.

precision()
sets the number of significant digits to write for floating-point values.

Manipulators

Use of the **setf()** , **width()** , and similar member functions is very convenient if the same specifications are used for a large number of inserted items. If the formatting frequently changes, it is more convenient to use a manipulator. A *manipulator* is an object that can be an operand to the << or >> operator, but which modifies the state of the stream, rather than actually inserting or extracting any data. For instance, the manipulators **hex** and **dec** can be used to request hexadecimal or decimal printing of integral values. Thus, the following sequence can be used to write out the value of **i** in decimal and the value of **x[i]** in hexadecimal:

```
cout << "i = " << dec << i << ",
   x[i] = " << hex << x[i];
```

Other manipulators include the following:

ws	skips white space on input.
flush	flushes a stream's buffer.
endl	inserts a newline character and then flushes the stream's buffer.
ends	inserts an end-of-string character (**'\0'**).

It is possible to create user-defined manipulators in addition to the standard manipulators, but this is beyond the scope of this book. See "class IOMANIP" on page 93 for examples of user-defined manipulators, or refer to the *C++ Programming Language, Second Edition* for a detailed discussion.

I/O Status Handling

Associated with each stream is a set of flags indicating the I/O state of the stream. For example, the flag **ios::eofbit** indicates that no more characters can be extracted from a stream, and the flag **ios::failbit** indicates that some previous request failed. The I/O state flags can be tested by member functions; for example, **cin.eof()** tests whether more characters can be extracted from the standard input stream. The I/O state flags can be individually manipulated by using the **clear()** member function; for example, **cout.clear(0)** clears all the I/O state flags for the stream **cout** .

For convenience, the **!** operator and the conversion to **void*** operator allow concise testing of a stream for any error. These operators allow you to use statements such as the following, which writes the results of the function **nextline** to the standard output stream until an error occurs:

```
while(cout)
   cout << nextline();
```

Because an insertion or extraction always produces its stream argument as its result, this can be further abbreviated to the following statement:

```
while(cout << nextline());
```

This form makes it more obvious that the loop might not terminate at all. Note that attempting to extract from a stream from which no more characters can be taken is considered a failure. Thus, you can use a loop such as the following to extract and process items from a stream until the stream is exhausted:

```
while(cin >> datum)
   process(datum);
```

370 I/O Considerations

Many C++ programs were developed under UNIX operating systems, which are where the first C++ translators were available. Unfortunately, I/O is very different under MVS and CMS than under UNIX. This may require you to make some changes to existing C++ applications to adapt to the mainframe environment. This section discusses some of these issues. For a more detailed discussion of the differences between 370 I/O and traditional UNIX I/O, and of portable I/O coding techniques, see the *SAS/C Library Reference, Volume 1.*

Text and Binary Access

Under UNIX and some other operating systems, files are stored as sequences of characters, and the newline character serves to separate one line from the next. Under MVS and CMS, files are stored as sequences of records, separated by gaps (which do not contain any characters). Files in such systems therefore support two forms of access: text and binary. When a file is accessed in binary mode, only the characters are accessed and information about lines or records is lost. When a file is accessed in text mode, record gaps are treated as if they contained a newline character. That is, a newline is inserted at the end of each record on input, and writing a newline on output causes a record gap to be generated, but no physical character is written.

Text access is generally more suitable for programs that need to be aware of line boundaries, while binary access is suitable for programs that must read or write data exactly as specified. (For example, a program that writes an object deck must use binary access, to keep newlines in the object deck from being translated to record gaps.)

When a stream is associated with an external file, the default access mode is text. You can specify **ios::binary** in the open mode if you need binary access, as in the following statement:

```
ostream object("tso:fred.obj",
               ios::out|ios::binary);
```

Filenames

Mainframe systems offer several unique ways of naming files, most of which bear little resemblance to traditional MYFILE.TXT style filenames found under UNIX. The general form of a file name is

style:name

where *style* is a code describing how the rest of the name is to be interpreted. The *style:* portion of a filename can be omitted, in which case a default is assumed.

MVS filenames

Under MVS, the following filename styles are supported:

ddn:*name*
indicates the *name* portion is a DDname, associated with a batch JCL DD statement or a TSO ALLOCATE

command. This style is the default MVS filename style; that is, a filename of XYZ is assumed to refer to DDname XYZ.

dsn:*name*
: indicates the *name* portion is a fully qualified data set name, for example, dsn:SYS1.MACLIB(DCB).

tso:*name*
: indicates the *name* portion is a data set name in TSO style. A fully qualified name is generated from it by using the prefix *name* with the TSO userid. For instance, opening tso:MY.DATA opens the file *userid*.MY.DATA, where *userid* is the userid of the user running the program. **tso** : style names most closely resemble typical filenames under UNIX systems.

CMS filenames

Under CMS, the following filename styles are supported:

cms:*name*
: indicates the *name* portion is a CMS filename, composed of a filename, a filetype, and an optional filemode, separated by blanks, for example, cms:PROFILE XEDIT. As a convenience, periods can be used in place of blanks to separate the portions of the filename, so that cms:PROFILE.XEDIT and cms:PROFILE XEDIT identify the same file. This style is the default CMS filename style.

sf:*fileid dirid*
: indicates the file is stored in the CMS Shared File System (SFS). The *fileid* specifies the name of a file and *dirid* specifies a directory path in the CMS Shared File System. For more information about this sort of filename, see the *SAS/C Compiler and Library User's Guide*.

ddn:*name*
: indicates the *name* portion is a DDname, defined using the FILEDEF command. This form of filename is useful mostly for compatibility with MVS.

You can change the default style for your program by including a declaration for an external **char*** variable named **_style** . For example, the following statement specifies that by default all filenames are to be interpreted as **tso:** style names:

```
char *_style = "tso:";
```

Amparms

To successfully and efficiently use a file on IBM 370 systems, it is often necessary to provide information about actual or desired file attributes in addition to the filename. For example, when you create a new file under MVS, it may be necessary to specify the file size and how large the records are to be. File attributes of this sort are specified via *amparms*, which are keyword parameters defined by the SAS/C library. Amparms can be specified as a third argument to an **fstream** constructor or **open()** member function. For example, the following statements specify that the DDname SYSLIN should define a file with fixed-length 80-byte records, grouped into blocks of 3,200 characters:

```
ostream objfile;
objfile.open("syslin", ios::out|ios::binary,
             "recfm=f,reclen=80,blksize=3200");
```

The translator supports the same amparms as the SAS/C Compiler, as described in the *SAS/C Library Reference, Volume 1*.

An additional optional argument to stream constructors and **open()** member functions is an access method name. For instance, the following statement defines an **iostream** processed by the "rel" access method.

```
iostream work("dsn:&wkfile",
              ios::out|ios:binary,
              "alcunit=cyl,space=5","rel");
```

See the *SAS/C Library Reference, Volume 1* for a discussion of the meaning and utility of specifying an access method.

File Positioning

Under UNIX systems, it is generally possible to reposition any file to a particular character, for example, the 10,000th character. Due to the inner workings of mainframe I/O, this is generally not possible under MVS or CMS. One exception is for files accessed for binary access, using the "rel" access method. These files can be freely positioned to particular characters in a way compatible with UNIX systems.

The streams library defines two different types to deal with file positioning, **streampos** and **streamoff** . The streampos type is used to hold file positions, while **streamoff** is used to hold offsets of one point in a file from another.

In UNIX implementations of C++, both **streampos** and **streamoff** are integral types and both may be used freely with any file. However, in the SAS/C implementation, only **streamoff** is an integral type. Further, **streamoff** can only be used for files that behave like UNIX files, namely, those opened for binary access using the "rel" access method. **streampos** is a nonintegral type that encodes a file position in a system-dependent manner. You can use the **tellg()** and **tellp()** member functions to obtain a **streampos** for a particular point in the file, and you can use the **seekg()** and **seekp()** functions to position to a point whose location has previously been determined. Because no arithmetic or other operations are defined on **streampos** values, more general positioning operations such as seeking 10,000 characters past a specific location in the file are not supported.

Other Differences between UNIX and 370 I/O

370 I/O is not based on file descriptors. For this reason, member functions such as **fd()** and **attach()** are not meaningful in the IBM 370 environment and are not supported. Similarly, UNIX protection modes are not meaningful to MVS or CMS and constructors or **open()**

calls that specify a protection mode will be rejected at compile time.

Compatibility Issues for C++ I/O

This section explains how the SAS/C implementation relates to releases of AT&T C++ and discusses some considerations for developing C++ programs that will adapt easily to the future directions of C++.

AT&T C++ Release 2

The streams library is fully compatible with the AT&T Release 2 streams library. One exception is that member functions that refer to file descriptors and to other aspects of U NIX low-level I/O, such as **`fd()`** and **`attach()`** , have not been implemented.

AT&T C++ Release 1

The C++ streams library was redefined after the first release, and the Release 2 version is somewhat incompatible with Release 1. (Release 1 is described in Stroustrop's book, *The C++ Programming Language, First Edition.*) The SAS/C C++ Development System implements the old Release 1 streams library for compatibility with old programs, if you include the old header file **`stream.h`**. However, this usage is obsolete and will probably be removed from the library at some point.

stream.h header file

The functions, constants, and types defined in the **`stream.h`** header file are provided for compatibility with older versions of some C++ I/O libraries and may not be supported by future versions of the SAS/C C++ I/O library.

All of the functions declared in this header file share an internal **`static`** data area and return a pointer to this data area. Each call to any of these functions overwrites this data area.

Here are the contents of the **`stream.h`** header file:

`char* form(char *format, . . .) ;`

provides **`printf`** style formatting of character strings. The **`format`** string is the same as the format string for the C **`printf`** function. (See the description of **`printf`** in the *SAS/C Library Reference, Volume 1* for more information.) The number and type of the arguments following the **`format`** argument are controlled by the **`format`** string.

```
char* oct(long value, int size = 0) ;
char* hex(long value, int size = 0) ;
char* dec(long value, int size = 0) ;
char* chr(long value, int size = 0) ;
char* str(char *value, int size = 0) ;
```

format **`value`** into a string and return a pointer to the string.

The following list describes each function in more detail:

`oct()`	formats **`value`** as an octal number using the digits 0-7.
`hex()`	formats **`value`** as a hexadecimal number using the digits 0-9 and uppercase A-F.
`dec()`	formats **`value`** as a decimal number using the digits 0-9.
`chr()`	formats **`value`** as a character.
`str()`	formats **`value`** as a string.

If **`size`** is zero, the returned string is exactly as long as needed to represent the value of **`1`** . Otherwise, if **`size`** is less than the length of the converted value, the converted value is truncated on the right. If **`size`** is greater than the length of the converted value, spaces are added to the left of the converted value.

```
istream& WS(istream& i) ;
void eatwhite(istream& i) ;
```

move the get pointer for **`i`** past any white space. If the current character of **`i`** is not a white space character, these functions do nothing. A white space character is any character for which the macro **`isspace`** returns true. See the *SAS/C Library Reference, Volume 1* for a description of **`isspace`** .

```
const int input = ios::in;
const int output = ios::out;
const int append = ios::app;
const int atend = ios::ate;
const int _good = ios::goodbit;
const int _bad = ios::badbit;
const int _fail = ios::failbit;
const int _eof = ios::eofbit;
```

are constants that are passed to or are returned from functions of the I/O library. These names are only provided for compatibility with old programs; the new **`ios::`** style names should be used in new programs. See "enum io_state" on page 70 for details on each constant.

`typedef ios::io_state state_value;`

provides the old name (**`state_value`**) for what is now defined as **`ios::io_state`** . See "enum io_state" on page 70 for details on **`ios::io_state`** .

Future Directions

ANSI Committee X3J16 is currently working on an official standard for C++, including the streams library. As this work proceeds, it is very likely that there will be additional changes in the specifications of the streams library, some of which may cause existing programs to fail. It is therefore recommended that C++ streams applications should be kept relatively simple, to lessen the chances of disturbance by redefinition of some of the more obscure parts of the library. Especially, we recommend that you avoid deriving from the **`streambuf`** classes, as this interface is infrequently used and is expected to be volatile.

Note: There is not yet an ANSI Standard for C++, but when one is developed, the SAS/C C++ Development System will comply with this Standard. At such time, ANSI compatibility will take precedence over compatibility with the C++ language described in *The C++ Programming Language*. △

I/O Class Descriptions

The class descriptions for the streams library are divided into three parts: description of stream classes (such as **istream** and **ostream**), description of the buffer classes (such as **filebuf** and **streambuf**), and description of manipulators (**class IOMANIP**).

Most C++ programmers need only understand the stream classes. However, if you are doing more advanced programming, such as creating your own streams, you may want to also read the descriptions of the buffer and manipulator classes.

The protected interfaces in each of the classes have been implemented in accordance with AT&T Version Release 3.0 but are beyond the scope of this book to describe. For information on the protected interface for these classes, refer to your C++ book (such as the *C++ Language System Release 3.0 Library Manual*).

Note: It is not recommended that you use the protected member functions in your applications. This interface is volatile and is quite likely to change when the ANSI committee generates a C++ standard. Using the protected interface in your applications makes them much more likely to be quickly obsolete. △

Stream Class Descriptions

This section provides descriptions of the stream classes, such as **iostream** and **strstream** . As noted previously, the protected interface to these functions is not documented in the descriptions that follow. Each class (or occasionally a set of related classes) is listed alphabetically. All class descriptions include the following information:

- a synopsis of the class
- a brief description of the purpose and structure of the class
- a discussion of parent classes
- a detailed description of the members of the class
- examples of using some of the members, if appropriate
- a SEE ALSO section that points you to related classes.

In addition, some class descriptions contain other sections, as appropriate for the class.

The following classes are described in this section:

Table 4.5

class fstream	**class strstream**
class ifstream	**class istrstream**
class ofstream	**class ostrstream**
class iostream	**class stdiostream**
class istream	**class ios**
class ostream	**class streampos**
class bsamstream	**class ibsamstream**
class obsamstream	

The **class ios** description is divided into several sections. One section describes the basic stream-manipulation functions. The other sections deal with the enumerations provided by **class ios** and the functions that manipulate these enumerations. Examples of the enumerations include the format flags, the I/O state flags, the open mode flags, and the **seek_dir** flags.

Note: The term *character* in the following class and function descriptions refers to either a **char** , a **signed char** , or an **unsigned char** . △

class bsamstream, ibsamstream, and obsamstream

Provide formatted File I/O Using BSAM Low-Level I/O

SYNOPSIS

```
#include <bsamstr.h>
class bsamstream : public iostream
{
public:
   bsamstream();
   bsamstream(const char *filename,
      int mode,
      const char *keywords = 0,
      int willseek = 0,
      bsam_exit_list *exit_list = 0);
   virtual ~bsamstream();
   void open(const char *filename,
      int mode,
      const char *keywords = 0,
      int willseek = 0,
      bsam_exit_list *exit_list = 0);
   int find(const char *name);
   int init_directory();
   int delete_member(const char *name);
   int rename_member
     (const char *old_name,
      const char *new_name);
   int stow(const char *name,
      char action = 'R',
      int user_data_length = 0,
      const void *user_data = 0,
      int alias = 0, int TT = 0,
      int R = 0, int TTRN = 0);
   int add_member(const char *name,
      int user_data_length = 0,
      const void *user_data = 0,
      int alias = 0, int TT = 0,
      int R = 0, int TTRN = 0);
   int replace_member(const char *name,
      int user_data_length = 0,
      const void *user_data = 0,
      int alias = 0, int TT = 0,
      int R = 0, int TTRN = 0);
   void close();
   void setbuf(char *buffer,
      size_t length);
   int error_info(bsambuf::error_id& id);
   void clear_error_info();
   void set_user_data(void *user_data);
   void *get_user_data();
   const char *get_ddname();
   const char *get_member();
   bsambuf *rdbuf();
   char dcbrecfm();
   short dcblrecl();
   short dcbblksize();
   DCB_t *getdcb();
} ;
class ibsamstream : public istream
{
public:
   ibsamstream();
   ibsamstream(const char *filename,
      int mode,
      const char *keywords = 0,
      int willseek = 0,
      bsam_exit_list *exit_list = 0);
   virtual ~ibsamstream();
   void open(const char *filename,
      int mode,
      const char *keywords = 0,
      int willseek = 0,
      bsam_exit_list *exit_list = 0);
   int find(const char *name);
   int init_directory();
   int delete_member(const char *name);
   int rename_member
     (const char *old_name,
      const char *new_name);
   int stow(const char *name,
      char action = 'R',
      int user_data_length = 0,
      const void *user_data = 0,
      int alias = 0, int TT = 0,
      int R = 0, int TTRN = 0);
   int add_member(const char *name,
      int user_data_length = 0,
      const void *user_data = 0,
      int alias = 0, int TT = 0,
      int R = 0, int TTRN = 0);
   int replace_member(const char *name,
      int user_data_length = 0,
      const void *user_data = 0,
      int alias = 0, int TT = 0,
```

```
      int R = 0, int TTRN = 0);
   void close();
   void setbuf(char *buffer,
      size_t length);
   int error_info(bsambuf::error_id& id);
   void clear_error_info();
   void set_user_data(void *user_data);
   void *get_user_data();
   const char *get_ddname();
   const char *get_member();
   bsambuf *rdbuf();
   char dcbrecfm();
   short dcblrecl();
   short dcbblksize();
   DCB_t *getdcb();
};
class ibsamstream : public istream
{
public:
   ibsamstream();
   ibsamstream(const char *filename,
      int mode,
      const char *keywords = 0,
      int willseek = 0,
      bsam_exit_list *exit_list = 0);
   virtual ~ibsamstream();
   void open(const char *filename,
      int mode,
      const char *keywords = 0,
      int willseek = 0,
      bsam_exit_list *exit_list = 0);
   int find(const char *name);
   int init_directory();
   int delete_member(const char *name);
   int rename_member
     (const char *old_name,
      const char *new_name);
   int stow(const char *name,
      char action = 'R',
      int user_data_length = 0,
      const void *user_data = 0,
      int alias = 0, int TT = 0,
      int R = 0, int TTRN = 0);
   int add_member(const char *name,
      int user_data_length = 0,
      const void *user_data = 0,
      int alias = 0, int TT = 0,
      int R = 0, int TTRN = 0);
   int replace_member(const char *name,
      int user_data_length = 0,
      const void *user_data = 0,
      int alias = 0, int TT = 0,
      int R = 0, int TTRN = 0);
   void close();
   void setbuf(char *buffer,
      size_t length);
   int error_info(bsambuf::error_id& id);
   void clear_error_info();
   void set_user_data(void *user_data);
   void *get_user_data();
   const char *get_ddname();
```

```
        const char *get_member();
        bsambuf *rdbuf();
        char dcbrecfm();
        short dcblrecl();
        short dcbblksize();
        DCB_t *getdcb();
    };
```

DESCRIPTION

These three classes, defined in the **<bsamstr.h>** header file, specialize the classes **iostream** , **istream** , and **ostream** for file I/O using a **bsambuf** . **class bsamstream** and **class obsamstream** define an identical set of functions. **class ibsamstream** omits the set of functions that can change the file.

PARENT CLASSES

class bsamstream inherits characteristics from **class iostream** . **class ibsamstream** inherits characteristics from **class istream** . **class obsamstream** inherits its characteristics from **class ostream** . All three of **class bsamstream** , **class ibsamstream** , and **class obsamstream** inherit characteristics from **class ios** . See the descriptions of these parent classes for details on functions and operators that are inherited.

CONSTRUCTORS

Each class defines two constructors.

bsamstream::bsamstream()
ibsamstream::ibsamstream()
obsamstream::obsamstream()

create an unopened stream of the appropriate type.

bsamstream::bsamstream
(const char *filename, int mode,
const char *keywords = 0,
int willseek = 0, bsam_exit_list
***exit_list = 0)**

ibsamstream::ibsamstream
(const char*filename, int mode,
const char *keywords = 0,
int willseek = 0,
bsam_exit_list *exit_list = 0)

ibsamstream::ibsamstream
(const char *filename, int mode,
const char *keywords = 0,
int willseek = 0,
bsam_exit_list *exit_list = 0)

create a stream of the appropriate type, using the DD name (and member name, if present) specified by **filename** , using the specified **mode** . The **ibsamstream** behaves as if **ios::in** was set in the **mode** argument, whether or not it was set by the caller. The **obsamstream** behaves as if **ios::out** was set in the **mode** argument, whether or not it was set by the caller.

All of the arguments are identical to the corresponding **bsambuf::open** arguments. Refer to “class bsambuf” on page 81 for a description of these arguments.

If the open fails, **ios::badbit** is set in the stream’s I/O state flags, as described in “enum io_state” on page 70 .

DESTRUCTORS

class bsamstream , **class ibsamstream** , and **class obsamstream** each have one destructor:

virtual bsamstream::~bsamstream()
virtual ibsamstream::~ibsamstream()
virtual obsamstream::~obsamstream()

close the stream, if opened. If the close fails, **ios::failbit** is set in the stream’s I/O state flags, as described in “enum io_state” on page 70 .

MEMBER FUNCTIONS

The following descriptions give the purpose and return type of the member functions, as well as any other appropriate information. Except where otherwise stated, the arguments have the same meaning, format, and use as the **bsambuf** arguments of the same name and function.

bsamstream::open(const char *filename,
int mode, const char *keywords = 0,
int willseek = 0,
bsam_exit_list *exit_list = 0)

ibsamstream::open(const char *filename,
int mode, const char *keywords = 0,
int willseek = 0,
bsam_exit_list *exit_list = 0)

obsamstream::open(const char *filename,
int mode, const char *keywords = 0,
int willseek = 0,
bsam_exit_list *exit_list = 0)

create a stream of the appropriate type, using the DDname (and member name, if present) specified by **filename** and using the specified **mode** . The **ibsamstream** constructor behaves as if **ios::in** was set in the **mode** argument, whether or not it was set by the caller. The **obsamstream** constructor behaves as if **ios::out** was set in the **mode** argument, whether or not it was set by the caller.

If the open fails, the **ios::badbit** is set in the stream’s I/O state flags, as described in “enum io_state” on page 70 .

int bsamstream::find(const char *name)
int ibsamstream::find(const char *name)
int obsamstream::find(const char *name)

position the file connected to the stream, which must be a PDS, to the start of the member identified by name. name may be specified in either upper- or lowercase. If **name** is shorter than eight characters, it will be padded on the right with blanks. **find** returns 0 if the file is successfully positioned or a value other than 0 otherwise.

int bsamstream::init_directory()
int obsamstream::init_directory()

invoke the "initialize directory" function of the STOW macro on the file (which must be a PDS) connected to the stream. The value returned is 0 if the function succeeds or is a value other than 0 otherwise.

`int bsamstream::delete_member`
`(const char *name)`

`int obsamstream::delete_member`
`(const char *name)`

deletes the PDS member identified by **`name`** from the file connected to the **`stream`** . **`name`** may be specified in either upper- or lowercase. If **`name`** is shorter than eight characters, it will be padded on the right with blanks. The value returned is 0 if the member is successfully deleted or is a value other than 0 otherwise.

`int bsamstream::rename_member`
`(const char *old_name,`
`const char *new_name)`

`int obsamstream::rename_member`
`(const char *old_name,`
`const char *new_name)`

renames the PDS member identified by **`old_name`** to **`new_name`** . **`old_name`** and **`new_name`** may be specified in either upper- or lowercase. If either name is shorter than eight characters, it will be padded on the right with blanks. The value returned is 0 if the member is successfully renamed or is a value other than 0 otherwise.

`int bsamstream::stow(const char *name,`
`char action = 'R',`
`int user_data_length = 0,`
`const void *user_data = 0, int alias = 0,`
`int TT = 0, int R = 0, int TTRN = 0)`

`int obsamstream::stow(const char *name,`
`char action = 'R',`
`int user_data_length = 0,`
`const void *user_data = 0, int alias = 0,`
`int TT = 0, int R = 0, int TTRN = 0)`

adds or replaces a member or alias in the file connected to the **`bsambuf`** . The file must be a PDS. **`stow`** returns a value other than 0 if any of the arguments are out of bounds. If **`stow`** invokes the STOW macro, with one exception, **`stow`** returns the return code from the STOW macro. (This code can also be retrieved via the **`error_info`** function.) In general, the STOW macro returns 0 if the requested action succeeded or a value other than 0 otherwise. The exception occurs when **`action`** is 'R' and the STOW macro returns 8. This return code from the STOW macro indicates that the member or alias did not previously exist and so was added. In this case, **`stow`** returns 0.

If an error occurs, the **`ios::badbit`** is set in the stream's I/O state.

`int bsamstream::add_member`
`(const char *name,`
`int user_data_length = 0,`
`const void *user_data = 0,`
`int alias = 0, int TT = 0,`
`int R = 0, int TTRN = 0);`

`int obsamstream::add_member`
`(const char *name,`
`int user_data_length = 0,`
`const void *user_data = 0,`
`int alias = 0, int TT = 0,`
`int R = 0, int TTRN = 0);`

is equivalent to the **`stow`** function when **`action`** is 'A'. The arguments and return value are identical to **`stow`** .

`int bsamstream::replace_member`
`(const char *name,`
`int user_data_length = 0,`
`const void *user_data = 0,`
`int alias = 0, int TT = 0,`
`int R = 0, int TTRN = 0)`

`int obsamstream::replace_member`
`(const char *name,`
`int user_data_length = 0,`
`const void *user_data = 0,`
`int alias = 0, int TT = 0,`
`int R = 0, int TTRN = 0)`

is equivalent to the **`stow`** function when **`action`** is 'R'. The arguments and return value are identical to **`stow`** .

`void bsamstream::close()`
`void ibsamstream::close()`
`void obsamstream::close()`

close the connection between the appropriate object and its associated file. Unless an error occurs, all bits in the object's I/O state are set to zero. If the close fails, **`ios::failbit`** is set in the stream's I/O state. The close could fail if the call to **`rdbuf()->close()`** fails. These functions have no return value.

`void bsamstream::setbuf(char *buffer,`
`size_t length)`

`void ibsamstream::setbuf(char *buffer,`
`size_t length)`

`void obsamstream::setbuf(char *buffer,`
`size_t length)`

call **`rdbuf()->setbuf(buffer, length)`** . These functions have no return value.

`bsambuf *bsamstream::rdbuf()`
`bsambuf *ibsamstream::rdbuf()`
`bsambuf *obsamstream::rdbuf()`

return a pointer to the **`bsambuf`** associated with the stream.

int bsamstream::error_info
(bsambuf::error_id& id)

int ibsamstream::error_info
(bsambuf::error_id& id)

int obsamstream::error_info
(bsambuf::error_id& id)
return the return code from the first failed low-level routine. See the table under the description of **bsambuf::error_info** for a list of values that can be stored in the **id** argument.

void bsamstream::clear_error_info()
void ibsamstream::clear_error_info()
void obsamstream::clear_error_info()
set the stored error information to the initial state. If **error_info** is called in this state, the value of **error_id** is **Enone** and the return code is 0.

void bsamstream::set_user_data
(void *user_data)

void ibsamstream::set_user_data
(void *user_data)

void obsamstream::set_user_data
(void *user_data)
stores the value in **user_data** for retrieval by **get_user_data** . The value may be any value required by the program. It is ignored by the stream.

void *bsamstream::get_user_data()
void *ibsamstream::get_user_data()
void *obsamstream::get_user_data()
retrieves the value stored by **set_user_data** .

void *bsamstream::get_ddname()
void *ibsamstream::get_ddname()
void *obsamstream::get_ddname()
returns a pointer to the DDname part of the filename of the associated file. The DDname is in uppercase characters.

void *bsamstream::get_member()
void *ibsamstream::get_member()
void *obsamstream::get_member()
returns a pointer to the member name part of the filename of the associated file. The member name is in uppercase characters. If the filename did not specify a member name, the pointer points to a 0-length string.

DCB_t *bsamstream::getdcb()
DCB_t *ibsamstream::getdcb()
DCB_t *obsamstream::getdcb()
return a pointer to the DCB associated with the stream.

char bsamstream::dcbrecfm()
char ibsamstream::dcbrecfm()
char obsamstream::dcbrecfm()
return the value of the DCBRECFM field in the DCB associated with the stream.

short bsamstream::dcblrecl()
short ibsamstream::dcblrecl()
short obsamstream::dcblrecl()
return the logical record length (LRECL) of the file connected to the stream.

short bsamstream::dcbblksize()
short ibsamstream::dcbblksize()
short obsamstream::dcbblksize()
return the physical block size (BLKSIZE) of the file connected to the stream.

SEE ALSO
class bsambuf, class bsam_exit_list

class fstream, ifstream, and ofstream

Provide Formatted File I/O

SYNOPSIS

```
#include <fstream.h>
class ifstream : public istream
{
public:
   ifstream();
   ifstream(const char *name,
            int mode = ios::in,
            const char *amparms = "",
            const char *am = "");
   virtual ~ifstream();
   void open(const char *name,
             int mode = ios::in,
             const char *amparms = "",
             const char *am = "");
   void close();
   void setbuf(char *p, int len);
   filebuf* rdbuf();
};
class ofstream : public ostream
{
public:
   ofstream();
   ofstream(const char *name,
            int mode = ios::out,
            const char *amparms = "",
            const char *am = "");
   virtual ~ofstream();
   void open(const char *name,
             int mode = ios::out,
             const char *amparms = "",
             const char *am = "");
   void close();
   void setbuf(char *p, int len);
   filebuf* rdbuf();
};
class fstream : public iostream
{
public:
   fstream();
   fstream(const char *name, int mode,
           const char *amparms = "",
           const char *am = "");
```

```
        virtual ~fstream();
        void open(const char *name, int mode,
                  const char *amparms = "",
                  const char *am = "");
        void close();
        void setbuf(char *p, int len);
        filebuf* rdbuf();
    };
```

DESCRIPTION

The three classes contained in the **fstream.h** header file specialize the classes **istream** , **ostream** , and **iostream** for file I/O. In other words, the streambuf associated with the I/O operations is a **filebuf** . The functions associated with each of the classes in this header file, **class ifstream** , **class ofstream** , and **class fstream** , are very similar.

PARENT CLASSES

class ifstream inherits characteristics from **class istream** . **class ofstream** inherits characteristics from **class ostream** . **class fstream** inherits characteristics from **class iostream** . All three of **class ifstream** , **ofstream** , and **fstream** inherit characteristics from **class ios** . See the descriptions of these parent classes for the details on functions and operators that are inherited.

CONSTRUCTORS

There are two sets of constructors for **class ifstream**, **class ofstream** , and **class fstream** , as follows:

ifstream::ifstream()
ofstream::ofstream()
fstream::fstream()

create an unopened stream of the appropriate type.

ifstream::ifstream(const char *name,
int mode = ios::in,
const char *amparms = " ",
const char *am = " ")

ofstream::ofstream(const char *name,
int mode = ios::out,
const char *amparms = "",
const char *am = "")

fstream::fstream(const char *name,
int mode, const char *amparms = "",
const char *am = "")

create a stream of the appropriate type, named **name** , using the specified **mode** . The **ifstream** constructor behaves as if **ios::in** was set in the **mode** argument, whether or not it was set by the caller. The **ofstream** constructor behaves as if **ios::out** was set in the **mode** argument, whether or not it was set by the caller.

An explanation of filename specification, as well as the arguments **amparms** and **am** can be found in the *SAS/C Library Reference, Volume 1.* (Note that the order of the **amparms** and **am** , arguments in these constructors is the opposite of the order in which they appear in calls to the C **afopen** function.) You may also want to refer to the *SAS/C Compiler and Library User's Guide*.

The available modes are described in "enum open_mode" on page 70 . If the open fails, the stream's status is reflected in its I/O state flags, as described in "enum io_state" on page 70 .

DESTRUCTORS

class ifstream , **class ofstream** , and **class fstream** each have one destructor: **virtual ifstream::~ifstream()**
virtual ofstream::~ofstream()
virtual fstream::~fstream()

close the stream, if opened.

MEMBER FUNCTIONS

The following descriptions give the purpose and return type of the member functions, as well as any other appropriate information.

void ifstream::open(const char *name,
int mode = ios::in,
const char *amparms = " ",
const char *am = " ")

void ofstream::open(const char *name,
int mode = ios::out,
const char *amparms = " ",
const char *am = " ")

void fstream::open(const char *name,
int mode,
const char *amparms = "",
const char *am = "")

open the named file using the specified **mode** . **ifstream::open()** behaves as if **ios::in** was set in the **mode** argument, whether or not it was set by the caller. **ofstream::open()** behaves as if **ios::out** was set in the **mode** argument, whether or not it was set by the caller. For **fstream** objects there is no default **open_mode** bit set.

The available open modes are described in "enum open_mode" on page 70 . An explanation of the arguments **amparms** and **am** can be found in the *SAS/C Library Reference, Volume 1.* (Note that the order of the **amparms** and **am** arguments in these functions is the opposite of the order in which they appear in calls to the C **afopen** function.) You may also want to refer to the *SAS/C Compiler and Library User's Guide*.

These functions do not have a return value. If an error occurs during the open, the **ios::failbit** bit is set in the stream's I/O state. Reasons for a failed open include that the file already exists, or that the call to **rdbuf()->open()** fails.

void ifstream::close()
void ofstream::close()
void fstream::close()

close the connection between the appropriate object and its associated file. Unless an error occurs, all bits in the object's I/O state are set to zero. The close could fail if the call to **rdbuf()->close()** fails. These functions have no return value.

void ifstream::setbuf(char *p, int len)
void ofstream::setbuf(char *p, int len)
void fstream::setbuf(char *p, int len)
call **filebuf::setbuf(p,len)** . These functions have no return value.

filebuf* ifstream::rdbuf()
filebuf* ofstream::rdbuf()
filebuf* fstream::rdbuf()
return a pointer to the **filebuf** associated with the stream.

SEE ALSO

```
class filebuf
```

class ios

Provide Buffer and Stream Manipulation

SYNOPSIS

```
#include <iostream.h>
class ios
{
public:
/* See the enum format_state, enum
   io_state, enum open_mode, and enum
   seek_dir descriptions for more
   definitions. */

   ios(streambuf *buf);
   virtual ~ios();

   int width();
   int width(int w);

   char fill();
   char fill(char c);

   int precision();
   int precision(int i);

   static unsigned long bitalloc();
   static int xalloc();
   long& iword(int i);
   void*& pword(int i);

   streambuf* rdbuf();

   ostream* tie();
   ostream* tie(ostream *s);
};
```

DESCRIPTION

The **iostream.h** header file declares **class ios** . This class and the classes derived from it provide an I/O interface for inserting information into and extracting information from **streambuf** . This I/O interface supports both formatted and unformatted information. This description is devoted to those operations used in stream and buffer manipulation.

Several enumerations are defined in **class ios** (**io_state**, **open_mode** , **seek_dir** , and the format flags). These enumerations are described in subsequent sections. The **open_mode** and **seek_dir** flags are not used directly by the functions in **class ios** but are used by classes derived from it.

PARENT CLASSES

class ios is the parent of all the stream classes. It has no parent class.

CONSTRUCTORS

class ios defines one constructor:

ios::ios(streambuf *buf)
sets up **buf** as the associated **streambuf** . If **buf** is **NULL** , the effect is undefined.

DESTRUCTORS

Here is the **class ios** destructor:

virtual ios::~ios()
closes the stream.

BUFFER AND STREAM MANIPULATION FUNCTIONS

class ios defines several functions that provide buffer and stream manipulation capabilities. The following list describes these functions.

streambuf* ios::rdbuf()
returns a pointer to the **streambuf** associated with the **ios** when it was created.

ostream* ios::tie()
returns the **ostream** currently tied to the **ios** , if any; returns **NULL** otherwise.

ostream* ios::tie(ostream *s)
ties **s** to the **ios** and returns the stream previously tied to this stream, if any; returns **NULL** otherwise.

If the **ios** is tied to an **ostream** , then the **ostream** is flushed before every read or write from the **ios** . By default, **cin** , **cerr** , and **clog** are tied to **cout** .

FORMATTING FUNCTIONS

class ios defines several functions that use and set the format flags and variables. **class ios** also provides functions you can use to define and manipulate your own formatting flags, plus several built-in manipulators that allow you to set various format flags.

Format flag functions

The following list describes some of the functions that use and set the library-supplied format flags. The **flags()** , **setf()** , and **unsetf()** functions and the format flags are described in “enum format_state” on page 67 .

int ios::width()
returns an **int** representing the value of the current field width.

int ios::width(int w)
sets the field width to **w** and returns an **int** representing the previous field width value.

The default field width is 0. When the field width is 0, inserters insert only as many characters as necessary to represent the value. When the field width is nonzero, inserters insert at least as many characters as are necessary to fill the field width. The fill character is used to pad the value, if necessary, in this case.

Numeric inserters do not truncate their values. Therefore, if the value being inserted is wider than the field width, the entire value is inserted, regardless of the field width overrun. As you can see, this implies that the field width value is a minimum constraint; there is no way to specify a maximum constraint on the number of characters to be inserted.

The field width variable is reset to 0 after each insertion or extraction. In this sense, the field width serves as a parameter for insertions and extractions.

You can also use the parameterized manipulator, **setw** , to set the field width.

char ios::fill()
returns a char representing the current fill character.

char ios::fill(char c)
sets the fill character to **c** and returns a **char** representing the previous value. The default fill character is a space. You can also set the fill character using the parameterized manipulator, **setfill** .

int ios::precision()
returns an **int** representing the current precision value.

int ios::precision(int i)
sets the precision to **i** and returns an **int** representing the previous precision. Use this function to control the number of significant digits included in floating-point values. The default precision is six. You can also set the precision using the parameterized manipulator, **setprecision** .

User-defined format flag functions
class ios includes four functions that you can use to define format flags and variables in addition to those described in “enum format_state” on page 67 .

static unsigned long ios::bitalloc()
returns an **unsigned long** with a single, previously unallocated, bit set. This allows you to create additional format flags. This function returns 0 if there are no more bits available. Once the bit is allocated, you can set it and clear it using the **flags()** , **setf()** , and **unsetf()** functions.

static int ios::xalloc()
returns an **int** that represents a previously unused index into an array of words available for use as format state variables. These state variables can then be used in your derived classes.

long& ios::iword(int i)
returns a reference to the **i** th user-defined word. **i** must be an index allocated by **ios::xalloc()** .

void*& ios::pword(int i)
returns a reference to the **i** th user-defined word. **i** must be an index allocated by **ios::xalloc()**. **pword()** is the same as **iword()** except that its return type is different.

Refer to the *C++ Language System Release 3.0 Library Manual* for details on defining and using user-defined format flags.

Built-in manipulators
Manipulators take an **ios&** , an **istream&** , or an **ostream&** and return their argument. The following built-in manipulators are useful with **ios** objects. **stream** has type **ios&** .

stream >> dec and **stream << dec**
set the conversion base for the stream to decimal (by setting the **ios::dec** bit and clearing **ios::oct** and **ios::hex**).

stream >> oct and **stream << oct**
set the conversion base for the stream to octal (by setting the **ios::oct bit** and clearing **ios::dec** and **ios::hex**).

stream >> hex and **stream << hex**
set the conversion base for the stream to hexadecimal (by setting the **ios::hex** bit and clearing **ios::dec** and **ios::oct**).

stream >> ws
extracts whitespace characters.

stream << endl
inserts a newline character and flushes the stream.

stream << ends
inserts a null (**\0**) character into the stream.

stream << flush
flushes the given **ostream** object.

In addition, parameterized manipulators are available to operate on **ios** objects. These include **setfill**, **setprecision** , **setiosflags** , and **resetiosflags** , described in detail in “class IOMANIP” on page 93 .

SEE ALSO

```
class iostream, class istream, class ostream
```

enum format_state

Provide Buffer and Stream Formatting

SYNOPSIS

```
#include <iostream.h>

class ios
{
```

```
public:

/* See the class ios, enum io_state,
   enum open_mode, and enum seek_dir
   descriptions for more definitions. */

   enum {skipws,

         left,
         right,
         internal,

         dec,
         oct,
         hex,

         showbase,
         showpoint,
         uppercase,
         showpos,

         scientific,
         fixed,

         unitbuf,
         stdio
        };
   static const unsigned long basefield;
   static const unsigned long adjustfield;
   static const unsigned long floatfield;

   unsigned long flags();
   unsigned long flags(unsigned long f);

   unsigned long setf(unsigned long mask);
   unsigned long setf(unsigned long
                      setbits, unsigned
                      long mask);
   unsigned long unsetf(unsigned long
                        mask);
};
```

DESCRIPTION

class ios (defined in the **iostream.h** header file) provides a format state, which is used by the stream classes to control formatting. The format state is controlled by the format flags, and **class ios** provides several functions to manipulate these flags. The member functions **flags()** , **setf()** , and **unsetf()** control the majority of formatting. These functions are described further on in this section. Other member functions that have an effect on the format state are **fill()**, **width()** , and **precision()** . These functions are described in the previous **class ios** description.

In addition to the predefined format flags, users can create their own user-defined format flags. This is described in the previous **class ios** description, under “User-defined format flag functions”.

FORMAT FLAGS

The following list describes each format flag in detail.

skipws
: skips whitespace on input. This flag applies only to scalar extractions. If **skipws** is not set, whitespace is not skipped.

To protect against looping, zero-width fields are considered a bad format. Therefore, if the next character is whitespace and **skipws** is not set, arithmetic extractors signal an error. **skipws** is set by default.

padding flags
: control the padding of formatted values. There are three of them:

left
: causes output to be left-adjusted.

right
: causes output to be right-adjusted. This is the default if none of the padding bits is set. **right** is set by default.

internal
: causes padding to occur between the sign or base indicator and the value.

These padding flags are grouped together by the member **adjustfield** . To set the fill character, use the **fill()** function. To control the width of formatted items, use the **width()** function.

conversion base flags
: control the conversion base of values, as follows:

dec	indicates the conversion base is decimal. This is the default for input if none of the conversion base flags is set. **dec** is set by default.
oct	indicates the conversion base is octal.
hex	indicates the conversion base is hexadecimal.

These conversion base flags are grouped together by the member **basefield** .

Although decimal is the default conversion base for insertions (if none of these flags is set), the default conversion base for extractions follows the C++ lexical conventions for integral constants. You can also use the built-in manipulators **dec**, **oct** , and **hex** to control the conversion base. These manipulators are described in the previous **class ios** description.

showbase
: causes the base indicator to be shown in the output. This form of output follows the C++ lexical conventions for integral constants. **showbase** is not set by default.

showpoint
: causes the output to include any trailing zeros and decimal points resulting from floating-point conversion. **showpoint** is not set by default.

`uppercase`
causes uppercase letters to be used in output. For example, an **`X`** is used instead of **`x`** in hexadecimal output, and an **`E`** is used instead of **`e`** in scientific notation. **`uppercase`** is not set by default.

`showpos`
causes a **`+`** sign to be added to the decimal conversion of positive integers. **`showpos`** is not set by default.

floating-point flags
control the format of floating-point conversions, as follows:

`scientific`
causes the value to be converted using scientific notation. In this form, there is one digit preceding the decimal point, and the number of digits after the decimal point is equal to the precision (set with the **`precision()`** function). The default precision is six. An **`e`** (or **`E`** if **`uppercase`** is set) precedes the exponent.

`fixed`
causes the value to be converted to decimal notation. The precision of the value is controlled with the **`precision()`** function. The default precision is six.

If neither **`scientific`** nor **`fixed`** is set, the value is converted to one or the other format, according to the following rules:

- □ If the exponent resulting from the conversion is less than -4 or greater than the precision, scientific notation is used.
- □ Otherwise, fixed notation is used.

Unless **`showpoint`** is set, trailing zeros are removed from the value, regardless of the format. A decimal point appears in the value only if it is followed by a digit. These flags are grouped together by the member **`floatfield`** . They are not set by default.

`unitbuf`
causes the stream to be flushed after an insertion. **`unitbuf`** is not set by default.

`stdio`
causes the standard C output files **`stdout`** and **`stderr`** to be flushed after an insertion. **`stdio`** is not set by default.

FORMATTING FUNCTIONS

The following functions can be used to turn format flags on and off.

`unsigned long ios::flags()`
returns an **`unsigned long`** representing the current format flags.

`unsigned long ios::flags`
`(unsigned long f)`
sets (turns on) all the format flags specified by **`f`** and returns an **`unsigned long`** representing the previous flag values.

`unsigned long ios::setf`
`(unsigned long mask)`
sets (turns on) only those format flags that are set in **`mask`** and returns an **`unsigned long`** representing the previous values of those flags. You can accomplish the same task by using the parameterized manipulator, **`setiosflags`** .

`unsigned long ios::setf`
`(unsigned long setbits,`
`unsigned long mask)`
turns on or off the flags marked by **`mask`** according to the corresponding values specified by **`setbits`** and returns an **`unsigned long`** representing the previous values of the bits specified by **`mask`** . The EXAMPLES section provides an example of using this function.

Using **`setf(0, mask)`** clears all the bits specified by **`field`** . You can accomplish the same task by using the parameterized manipulator **`resetiosflags`** .

`unsigned long ios::unsetf`
`(unsigned long mask)`
clears the format flags specified by **`mask`** and returns an **`unsigned long`** representing the previous flag values.

EXAMPLES

The **`setf()`** function is used to change format flags. For example, if you want to change the conversion base in an **`ios`** object called **`s`** , you could use the following expression:

```
s.setf(ios::hex, ios::basefield)
```

In this example, **`ios::basefield`** represents the conversion base bits you want to change, and **`ios::hex`** is the new value.

To set a flag that is not part of a field, use **`setf()`** with a single argument, as in the following example, which sets the **`skipws`** flag:

```
s.setf(ios::skipws)
```

To clear the **`skipws`** flag, use **`unsetf()`** :

```
s.unsetf(ios::skipws)
```

As another example of using **`setf()`** , suppose you want to clear in your **`ios`** object **`s`** all the bits specified by the variable **`clearbits`** . You could use the following expression to accomplish this:

```
s.setf(0, clearbits)
```

enum io_state

Provide Stream I/O State

SYNOPSIS

```
#include <iostream.h>

class ios
{
public:
   enum io_state {goodbit = 0,
                  eofbit,
                  failbit,
                  badbit
                 };

/* See the class ios, enum format_state,
   enum open_mode, and enum seek_dir
   descriptions for more definitions. */

   int rdstate();
   int eof();
   int fail();
   int bad();
   int good();
   void clear(int i = 0);

   operator void*();
   int operator !();
};
```

DESCRIPTION

class ios (defined in the **iostream.h** header file) defines **io_state** flags that represent the internal state of an **ios** object. Each flag has a value that can be set or reset independently for an **ios** object. Note that **goodbit** is not a flag but rather a symbolic name for the condition in which no flags are set. The functions such as **rdstate()** and **eof()** use and manipulate the I/O state flags.

I/O STATE FLAGS

A stream is in an unusual state (error or EOF) if any of the I/O state flags are set for the stream. If none of the flags are set, the stream is in the normal (nonerror) state. The meanings of the **io_state** enumerators are as follows:

goodbit	is not a flag. It is a symbolic name for the condition in which no flags are set.
eofbit	indicates the end of file has been encountered. If the stream is repositioned after **eofbit** is set, the bit is cleared.
failbit	indicates an error other than an I/O error, such as an error in formatting. Once the **failbit** bit is cleared, I/O can usually continue. **failbit** is also set if an operator or member function fails because no more characters can be extracted.
badbit	indicates an I/O operation failed. Do not continue I/O operations after this bit is set.

I/O STATE FUNCTIONS

class ios also provides several functions that use or manipulate the I/O state flags. In addition to the I/O state functions, **class ios** also defines two operators that allow you to check the I/O state of an **ios** object.

The following functions use and manipulate the values of the I/O state flags.

int ios::rdstate()
returns the current I/O state.

int ios::eof()
returns the value of **eofbit** if **eofbit** is set; otherwise, returns 0.

int ios::fail()
returns the value of **failbit** if **failbit** is set; otherwise, returns 0.

int ios::bad()
returns the value of **badbit** if **badbit** is set; otherwise, returns 0.

int ios::good()
returns a nonzero value if no bits are set in the stream's I/O state; otherwise, returns 0.

void ios::clear(int i = 0)
sets the stream's I/O state to **i** . The default value for **i** is 0. **clear()** has no return value.

The following two operators are useful when checking the I/O state of an **ios** object.

ios::operator void*()
converts an **ios** object to a pointer. If no bits are set in the stream's I/O state, this operator returns a pointer value that is not null. If **failbit** or **badbit** is set, the operator returns 0.

int ios::operator !()
converts an **ios** object to 0 if no bits are set in the stream's I/O state, or to a nonzero value if any bits are set in the stream's I/O state.

enum open_mode

Provide Buffer and Stream Open Modes

SYNOPSIS

```
#include <iostream.h>

class ios
{
public:
/* See the class ios, enum format_state,
   enum io_state, and enum seek_dir
   descriptions for more definitions. */

   enum open_mode {in,
                   out,
                   ate,
                   app,
                   trunc,
                   nocreate,
                   noreplace,
                   binary
                  };
};
```

DESCRIPTION

The **open_mode** enumeration is defined in **iostream.h**. This enumeration defines a number of flags that can be used when creating or opening a stream to specify attributes of the stream. You can specify several attributes simultaneously by using the OR operator to combine them. For example, to specify both the **out** and **binary** flags, use **ios::out|ios::binary** .

Only the **ios::ate** and **ios::app** flags are meaningful for string streams, such as **strstream** objects. See the description of **class strstream** , **class istrstream** , and **class ostrstream** for information on how these flags are used with these classes.

The following list describes the meaning of the **open_mode** flags for the file-oriented stream classes:

in
: means access the file for input.

out
: means access the file for output. If the file already exists, it is truncated unless one of **ios::in** , **ios::ate** , or **ios::app** is also specified.

ate
: means to position the file to the end of the file when the file is opened.

app
: means to access the file in append mode. In append mode, each output operation to the file causes the file to be positioned to the end before writing.

trunc
: means to truncate the file (making it empty) when it is opened. **ios::trunc** has no effect if the file does not yet exist.

nocreate
: means the open fails if the file to be opened does not exist.

noreplace
: means the open fails if the file already exists.

binary
: means to access the file in binary mode. If **ios::binary** is not specified, the file is accessed in text mode. See the *SAS/C Library Reference, Volume 1* for more information on the differences between text mode and binary mode.

enum seek_dir

Provide Buffer and Stream Seeking

SYNOPSIS

```
#include <iostream.h>

class ios
{
public:

/* See the class ios, enum format_state,
   enum open_mode, and enum open_mode
   descriptions for more definitions. */

   enum seek_dir {beg, cur, end};
};
```

DESCRIPTION

When you perform a seek on a stream, you must specify the starting point for the seek. **class ios** (defined in the **iostream.h** header file) provides the **seek_dir** flags to control seeking. The following list describes these flags:

beg	means the seek is relative to the beginning of the stream.
cur	means the seek is relative to the current position of the stream.
end	means the seek is relative to the end of the stream.

class iostream

Provide Bidirectional Stream

SYNOPSIS

```
#include <iostream.h>
class iostream : public ostream,
                 public istream
{
public:
   iostream(streambuf *buf);
};
```

DESCRIPTION

The **iostream.h** header file also provides **class iostream** , which is both an **istream** and an **ostream.** **class iostream** includes all the operations of both subclasses. It adds only a constructor of its own.

PARENT CLASSES

class iostream inherits characteristics from both **class istream** and **class ostream** . See the descriptions of these parent classes for the details on functions and operators that are inherited.

CONSTRUCTORS

class iostream defines one constructor:

iostream::iostream(streambuf *buf)
: sets up **buf** as the associated **streambuf** . If **buf** is **NULL** , the effect is undefined.

SEE ALSO

class ios , **class istream** , **class ostream**

class istream

Provide for Stream Extraction

SYNOPSIS

```
#include <iostream.h>

class istream : virtual public ios
{
public:
   istream(streambuf *buf);

   virtual ~istream();

   int ipfx(int need = 0);

   istream& operator >>(char *str);
   istream& operator >>(unsigned char *str);
   istream& operator >>(signed char *str);

   istream& operator >>(char& c);
   istream& operator >>(unsigned char& c);
   istream& operator >>(signed char& c);

   istream& operator >>(short& sh);
   istream& operator >>(unsigned short& sh);

   istream& operator >>(int& i);
   istream& operator >>(unsigned int& i);

   istream& operator >>(long& l);
   istream& operator >>(unsigned long& l);

   istream& operator >>(float& f);
   istream& operator >>(double& d);
   istream& operator >>(long double& ld);

   istream& operator >>(streambuf *buf);

   istream& operator >>(istream&(*f)
                        (istream&));
   istream& operator >>(ios&(*f)(ios&));

   istream& get(char *str, int len,
                char delim = '\n');
   istream& get(unsigned char *str,
                int len,
                char delim = '\n');
   istream& get(signed char *str,
                int len,
                char delim = '\n');
   istream& get(signed char& c);
   istream& get(unsigned char& c);
   istream& get(char& c);
   istream& get(streambuf& sb,
                char delim = '\n');

   int get();

   istream& getline(char *str, int len,
                    char delim = '\n');
   istream& getline(unsigned char *str,
                    int len,
                    char delim = '\n');

   istream& getline(signed char *str,
                    int len,
                    char delim = '\n');

   istream& ignore(int n = 1,
                   int delim = EOF);

   istream& read(char *str, int n);
   istream& read(unsigned char *str,
                 int n);
   istream& read(signed char *str,
                 int n);

   int gcount();
   int peek();

   istream& putback(char c);
   int sync();

   istream& seekg(streampos pos);
   istream& seekg(streamoff offset,
                  seek_dir place);

   streampos tellg();
};

istream& ws(istream&);
```

DESCRIPTION

class istream is defined in the **iostream.h** header file and is the base class for those stream classes that support only input. It includes all the basic extraction functions (formatted input) on fundamental C++ types, as well as a number of unformatted input functions and several functions that enable you to move the get pointer. It also includes one manipulator. These members are described in the following sections.

PARENT CLASSES

class istream inherits characteristics from **class ios** . See the description of this parent class for the details on functions and operators that are inherited.

CONSTRUCTORS

class istream defines one constructor:

istream::istream(streambuf *buf)
initializes an **istream** and associates a **streambuf** with it.

DESTRUCTORS

Here is the **class istream** destructor:

virtual istream::~istream()
closes the **istream** .

INPUT PREFIX FUNCTION

class istream defines an input prefix function that performs those operations that must be done before each formatting operation. This function is defined as follows:

```
int istream::ipfx(int need = 0);
```

If any I/O state flags are set for the **istream** , this function returns 0 immediately. If necessary, it flushes the **ios** (if any) tied to this **istream** . Flushing is necessary if **need** is 0 or if there are less than **need** characters available for input.

If **ios::skipws** is set and **need** is 0, then this function causes any leading white space in the input to be skipped. If an error occurs during this skipping, **ipfx()** returns 0. If no errors have occurred, this function returns 1.

This function is called by all formatted extraction operations and should be called by user-defined extraction operators unless the first input operation used by the user-defined extraction operator is a formatted extraction. For user-defined operations, **ipfx()** should be called with the argument equal to 0.

FORMATTED INPUT FUNCTIONS

The functions named **operator >>** are called extraction operators. They are formatted input functions. They each call the input prefix function **ipfx(0)** and do nothing else if it returns 0. If **ipfx(0)** does not return 0, the formatted input functions extract leading characters from the associated **streambuf** according to the type of their argument and the formatting flags in the **ios** . They all return the address of the **istream** .

Errors during extraction are indicated by setting the appropriate I/O state flags for the stream, as follows:

ios::failbit
means that the actual input characters did not match the expected input format.

ios::badbit
means that an error occurred during extraction of characters from the **streambuf** .

Here are the functions.

istream& istream::operator >>(char *str)

istream& istream::operator >>(unsigned char *str)

istream& istream::operator >>(signed char *str)
extract characters up to the next white space character. The terminating white space character is not extracted. If **width()** is nonzero, these functions extract no more than **width() - 1** characters and reset **width()** to 0. These functions add a terminating null character, even if an error occurs during extraction.

istream& istream::operator >>(char& c)

istream& istream::operator >>(unsigned char& c)

istream& istream::operator >>(signed char& c)
extract a single character and store it in the argument.

istream& istream::operator >>(short& sh)

istream& istream::operator >>(unsigned short& sh)

istream& istream::operator >>(int& i)

istream& istream::operator >>(unsigned int& i)

istream& istream::operator >>(long& l)

istream& istream::operator >>(unsigned long& l)
extract a number and store it in the argument. There may be a leading sign character (+ or -). If any of **ios::dec** , **ios::oct** , or **ios::hex** is set in the formatting state, characters are extracted and converted according to the bit that is set. If none of these bits is set, then these functions expect any of the following formats:

0x *hhh*

0X*hhh*

0*ooo*

ddd

Extraction stops when it reaches an unacceptable character. The acceptable characters are

0-7
for octal conversion

0-9
for decimal conversion

0-9, a-f, and A-F
for hexadecimal conversion.

ios::failbit is set if no digits are found.

istream& istream::operator >>(float& f)

istream& istream::operator >>(double& d)

istream& istream::operator >>(long double& ld)
extract a floating-point number and store it in the argument. The expected input format is an optional sign, followed by a decimal mantissa (optionally including a decimal point), followed by an optional floating-point exponent. The exponent may contain either an uppercase or a lowercase E and may have a + or - following the E. Extraction stops when EOF is encountered, or when a

character is read that cannot continue the previous input in a valid manner. **ios::failbit** is set if there are no digits to extract or if the format is not correct.

istream& istream::operator >>(streambuf *buf)
extracts all characters from the **istream** and inserts them into the **streambuf** . Extraction stops when no more characters can be obtained from the **istream** .

istream& istream::operator >> (istream& (*f) (istream&))

istream& istream::operator >> (ios& (*f) (ios&))
are for support of simple manipulators. Although these operators resemble an extraction in appearance, they are used to manipulate the stream rather than to extract characters from it. The argument to either of these operators is a manipulator function that modifies its **ios** or **istream** argument in some manner.

UNFORMATTED INPUT FUNCTIONS

The following functions are the unformatted input functions. They each call **ipfx(1)** first and do nothing else if 0 is returned.

istream& istream::get(char *str, int len, char delim = '\n')

istream& istream::get(unsigned char *str, int len, char delim = '\n')

istream& istream::get(signed char *str, int len, char delim = '\n')
extract up to **len - 1** characters. Extraction stops when a **delim** character is extracted, when no more characters are available, or when **len - 1** characters have been found. These functions store a terminating null character in the array. **ios::failbit** is set only if no characters at all were extracted.

istream& istream::getline(char *str, int len, char delim = '\n')

istream& istream::getline (unsigned char *str, int len, char delim = '\n')

istream& istream::getline (signed char *str, int len, char delim = '\n')
behave like the **get()** functions, except that the terminating **delim** character (if found) is extracted. A terminating null character is always stored in the array.

istream& istream::get(streambuf& sb, char delim = '\n')
extracts characters up to the next delim character or EOF and inserts them into **sb** . **delim** is not extracted or inserted. **ios::failbit** is set if an error occurs while inserting into **sb** .

istream& istream::get(signed char& c)
istream& istream::get(unsigned char& c)
istream& istream::get(char& c)
extract a single character. **ios::failbit** is set if no characters can be extracted.

int istream::get()
extracts a single character and returns it. EOF is returned if no characters can be extracted. **ios::failbit** is never set.

istream& istream::ignore(int n = 1, int delim = EOF)
extracts up to the next **n** characters or up to the next **delim** character. **ios::failbit** is never set.

istream& istream::read(char *str, int n)

istream& istream::read(unsigned char *str, int n)

istream& istream::read(signed char *str, int n)
extract the next **n** characters and store them into the array pointed to by **s** .

```
ios::failbit
```

is set if fewer than **n** characters can be extracted.

OTHER MEMBER FUNCTIONS

class istream includes several other functions, as follows:

int istream::gcount()
returns the number of characters extracted by the last unformatted extraction function. Formatted extraction functions may change the value of this function in unexpected ways.

int istream::peek()
returns EOF if **ipfx(1)** returns 0 or if no characters remain to be extracted. Otherwise it returns the next character in the stream without extracting it.

istream& istream::putback(char c)
does nothing if any bits are set in the stream's I/O state. If no bits are set in the stream's I/O state, this function pushes back the character **c** so it will be the next character extracted. **c** must be the same as the last character extracted from the istream. **ios::badbit** is set if the streambuf cannot push **c** back.

int istream::sync()
calls **sync()** on the associated **streambuf** . This function returns whatever the **streambuf::sync()** call returned.

istream& istream::seekg(streampos pos)

istream& istream::seekg(streamoff offset, seek_dir place)
move the get pointer of the associated **stream-buf.pos** is a value returned by a previous call to tellg(). offset and place are explained in the **streambuf::seekoff()** .

streampos istream::tellg()
returns the current **streampos** of the get pointer of the associated **streambuf** .

MANIPULATORS

The following function is a manipulator. It is intended to be used with the extractors to manipulate the stream in a specified way. This manipulator does nothing if any of the stream's I/O state flags are set. It signals an error by setting flags in the stream's I/O state. It returns its argument.

istream& ws(istream&);
skips over any white space in the stream.

SEE ALSO

```
class ios, class iostream, class ostream
```

class ostream

Provide for Stream Insertion

SYNOPSIS

```
#include <iostream.h>

class ostream : virtual public ios
{
public:
   ostream(streambuf *buf);
   virtual ~ostream();

   int opfx();

   void osfx();

   ostream& operator <<(char c);
   ostream& operator <<(signed char c);
   ostream& operator <<(unsigned char c);

   ostream& operator <<(const char *str);
   ostream& operator <<(const unsigned
                        char *str);
   ostream& operator <<(const signed
                        char *str);

   ostream& operator <<(short sh);
   ostream& operator <<(unsigned short sh);

   ostream& operator <<(int i);
   ostream& operator <<(unsigned int i);

   ostream& operator <<(long l);
   ostream& operator <<(unsigned long l);

   ostream& operator <<(float f);
   ostream& operator <<(double d);

   ostream& operator <<(void *vp);

   ostream& operator <<(streambuf *buf);

   ostream& operator <<(ostream&(*f)
                        (ostream&));
   ostream& operator <<(ios&(*f)(ios&));

   ostream& put(char c);

   ostream& write(const char *str, int n);
   ostream& write(const signed char *str,
               int n);
   ostream& write(const unsigned char *str,
               int n);
   ostream& flush();

   streampos tellp();

   ostream& seekp(streampos pos);
   ostream& seekp(streamoff offset,
               seek_dir place);
};
ostream& endl(ostream&);
ostream& ends(ostream&);
ostream& flush(ostream&);
```

DESCRIPTION

class ostream is declared in the **iostream.h** header file and is the base class for those classes that support only output. It includes all the basic insertion operators (formatted output) on fundamental C++ types, as well as a number of unformatted output functions and functions designed to change the stream position. In addition, some output manipulators are defined for use with this class.

PARENT CLASSES

class ostream inherits characteristics from **class ios** . See the description of this parent class for the details on functions and operators that are inherited.

CONSTRUCTORS

class ostream defines one constructor:

ostream::ostream(streambuf *buf)
initializes an **ostream** and associates a **stream-buf** with it.

DESTRUCTORS

Here is the **class ostream** destructor:

virtual ostream::~ostream()
closes the ostream.

PREFIX AND SUFFIX OUTPUT FUNCTIONS

Certain operations are defined to happen either before or after formatted output through a **ostream** . The prefix operations are done by **ostream::opfx()** and the suffix operations are done by **ostream::osfx()** .

int ostream::opfx()
performs prefix operations for an **ostream** . This function returns 0 and does nothing else if any bits in the stream's I/O state are set; it returns 1 otherwise. If the **ostream** is tied (see **tie()**) to another, the other stream is flushed (see **flush()**).

By convention, **opfx()** is called before any formatted output operation on a stream. If it returns 0 (meaning one or more bits are set in the stream's I/O state), the output operation is not performed. Each of the built-in inserters follows this convention. User-defined formatted output functions should also follow this convention by calling this function and checking the return code before doing any output.

void ostream::osfx()
performs suffix operations on the stream. If **ios::unitbuf** is set, this **ostream** is flushed. If **ios::stdio** is set, **cout** and **cerr** are flushed. This function should be called at the end of any formatted output function that does unformatted output on the **ostream** . It need not be called if the last output operation on the **ostream** was formatted.

FORMATTED OUTPUT FUNCTIONS

The functions named **operator <<** are called inserters (because they insert values into the output stream). All inserters are formatted output operations and as such follow the formatted output conventions mentioned previously.

All of the inserters do the following: First, they call **opfx()** , and if it returns 0, they do nothing else. If there is no error, they then convert the input argument to a converted value (a sequence of characters), based on the argument's type and value and on the formatting flags set for the stream. The rules for construction of the converted value are given here for each inserter.

Once a converted value has been determined, it is copied, possibly with the addition of fill characters, to an output field. The characters of the output field are then inserted into the stream's buffer. The **ios::width()** function for the stream determines the minimum number of characters in the output field. If the converted value had fewer characters, fill characters (defined by the value of **ios::fill()** for the stream) are added to pad out the field. The placement of fill characters is as follows::

ios::right
places the converted value in the rightmost portion of the field (leading padding).

ios::left
places the converted value in the leftmost portion of the field (trailing padding).

ios::internal
places the sign and base indicators of the converted value in the leftmost portion of the field and the remainder in the rightmost portion (internal padding).

Note that truncation cannot occur when copying the converted value to an output field, regardless of the value of **width()** .

Once the converted value is constructed and the field is padded to be at least **ios::width()** characters wide, **ios::width()** is reset to 0 and **osfx()** is called. All inserters indicate errors by setting I/O state flags in the **ostream** , as necessary. Inserters always return a reference to their **ostream** argument.

Here are the formatted output functions.

ostream& ostream::operator <<(char c)

ostream& ostream::operator <<(signed char c)

ostream& ostream::operator <<(unsigned char c)
convert the argument to the **char c** .

ostream& ostream::operator <<(const char *str)

ostream& ostream::operator <<(const unsigned char *str)

ostream& ostream::operator <<(const signed char *str)
convert the argument to a sequence of **chars** , up to but not including a ' **\0** ' character, pointed to by **str** .

ostream& ostream::operator <<(short sh)

ostream& ostream::operator <<(unsigned short sh)

ostream& ostream::operator <<(int i)

ostream& ostream::operator <<(unsigned int i)

ostream& ostream::operator <<(long l)

ostream& ostream::operator <<(unsigned long l)
convert the value of the argument to a sequence of digits, preceded by a leading ' **-** ' if the argument is negative.

If the following format flags within the **ostream** are set, they affect the converted value as follows:

ios::showpos
causes a leading ' **+** ' to be included in the converted value if the value is positive.

ios::dec , , **ios::oct** , and **ios::hex**
determine the base used for the converted value.

`ios::showbase`
causes the converted value to indicate the appropriate base as follows:

`decimal`	makes no change to the converted value.
`octal`	prefixes the converted value with a single ' **`0`** ' digit. If the value is 0, there is only one zero digit.
`hexa-decimal`	prefixes the converted value with ' **`0x`** '. If **`ios::uppercase`** is set, a leading ' **`0X`** ' is used instead.

If both a sign representation (+ or -) and a base representation appear in the converted value, the sign appears first.

`ostream& ostream::operator <<(float f)`

`ostream& ostream::operator << (double d)`
convert the argument to a character representation of its value in one of two formats:

- □ fixed notation (' ± ***ddd.ddd*** ')
- □ scientific notation (' ± ***d.ddde± dd*** ').

These formats are described in detail in "enum format_state" on page 67 . The format of the converted value is affected by the settings of the following format flags:

`ios::fixed` or **`ios::scientific`**
determines the overall representation format. If neither is set, then the overall format is scientific if the exponent is less then -4 or greater than the precision. Fixed notation is chosen otherwise.

`ios::showpoint`
causes the decimal point to be shown, followed by at least one digit. If **`showpoint`** is not set and all digits after the decimal point are zero, these digits and the decimal point are dropped.

`ios::uppercase`
causes the ' **`e`** ' in scientific notation to be ' **`E`** ' instead.

`ios::showpos`
causes a leading '+' to be output for positive values.

`ostream& ostream::operator << (void *vp)`
converts the value of the pointer **`vp`** to an **`unsigned long`** and represents it as if **`ios::hex`** and **`ios::showbase`** were set.

`ostream& ostream::operator << (streambuf *buf)`
fetches all the characters in **`buf`** and inserts them into the output stream, provided no bits are set in **`buf`** 's I/O state. No padding is done. If any bits are set in **`buf`** 's I/O state, this function returns immediately.

`ostream& ostream::operator << (ostream&(*f) (ostream&))`

`ostream& ostream::operator << (ios&(*f)(ios&))`
are for support of simple manipulators. Although these operators resemble an insertion in appearance, they are used to manipulate the stream rather than to insert characters into it. The argument to either of these operators is a manipulator function that modifies its **`ios`** or **`ostream`** argument in some manner.

UNFORMATTED OUTPUT FUNCTIONS

The following functions are for support of unformatted output to a stream. Because they are unformatted operations they do not call **`opfx()`** and **`osfx()`** ; however, these functions do check to see whether any I/O state flags are set for the **`ostream`** and take no further action if any are found. All inserters indicate errors by setting I/O state flags in the **`ostream`** . Each of these functions returns a reference to its argument **`ostream`** .

`ostream& ostream::put(char c)`
inserts its argument into the stream.

`ostream& ostream::write(const char *str, int n)`

`ostream& ostream::write(const signed char *str, int n)`

`ostream& ostream::write(const unsigned char *str, int n)`
insert **`n`** characters starting at **`str`** into the stream. The characters are treated as plain **`chars`** independent of their actual type. The null character is treated the same as any other character.

OTHER MEMBER FUNCTIONS

The following functions are also members of **`class ostream`** .

`ostream& ostream::flush()`
calls **`rdbuf()->sync()`** . For more information, refer to the description of **`streambuf::sync()`** .

`streampos ostream::tellp()`
returns the stream's current put pointer position. For more information, refer to the descriptions of **`streambuf::seekoff()`** and **`streambuf::seekpos()`** .

ostream& ostream::seekp(streampos pos)

ostream& ostream::seekp(streamoff offset, seek_dir place)
reposition the stream's put pointer. For more information, refer to the descriptions of **streambuf::seekoff()** and **streambuf::seekpos()** .

MANIPULATORS

The following functions are called manipulators. They are intended to be used with the inserters to manipulate the stream in specified ways. These manipulators do nothing if any of the stream's I/O state flags are set. They signal errors by setting flags in the stream's I/O state. They each return their argument.

ostream& endl(ostream&)
inserts a ' **\n** ' character into the stream. Here is an example:

```
#include <iostream.h>

float mynum=3.2;
   // Writes "mynum is:" and the value
      of mynum on one line.
cout << "mynum is: " << mynum << endl;
```

ostream& ends(ostream&)
inserts a ' **\0** ' character into the stream. Here is an example, using a **strstream** :

```
#include <iostream.h>
strstream mystream;
float mynum=3.2;

   // Writes mynum to mystream.
mystream << "mynum is: " << mynum <<
   ends;
```

ostream& flush(ostream&)
calls **ostream.flush()** .

SEE ALSO

```
class ios, class iostream, class istream
```

class stdiostream

Provide Formatted I/O in a Mixed C and C++ Environment

SYNOPSIS

```
#include <stdiostream.h>

class stdiostream : public iostream
{
public:
   stdiostream(FILE *file);

   FILE* stdiofile();

   stdiobuf* rdbuf();
};
```

DESCRIPTION

class stdiostream is declared in the **stdiostream.h** header file. It provides **iostream** access to an external file accessed by C functions using the ANSI standard I/O interfaces declared in **stdio.h** . Use of **class stdiostream** enables a program to use stdio output and C++ **iostream** output in the same output file. Similarly, **class stdiostream** enables your program to use **stdio** input and C++ **iostream** input to process the same input file.

PARENT CLASSES

class stdiostream inherits characteristics from **class iostream** , which in turn inherits characteristics from **class istream** , **class ostream** , and **class ios** . See the descriptions of these parent classes for the details on functions and operators that are inherited.

CONSTRUCTORS

class stdiostream has one constructor:

stdiostream::stdiostream(FILE *file)
creates a stream from the open **FILE* file** . The constructor assumes that the file is open.

MEMBER FUNCTIONS

The following descriptions give the purpose and return type of the member functions, as well as any other appropriate information.

stdiostream::FILE* stdiofile()
returns the **FILE*** associated with this stream.

stdiobuf* stdiostream::rdbuf()
returns a pointer to the **stdiobuf** associated with the stream.

SEE ALSO

```
class stdiobuf
```

class streampos

Mark a Stream Location

SYNOPSIS

```
#include <iostream.h>

class streampos
{
public:
   streampos();
   streampos(long n);
   operator long();
   fpos_t* fpos();
};
```

DESCRIPTION

class streampos is declared in the **iostream.h** header file and is used to record or specify a position in a stream. This class is for use only with streams that support seeking.

Most **streampos** values are similar to C **fpos_t** values; that is, they record file position values in a way private to the implementation. Because these values are probably not useful for user-defined stream classes, it is also possible to create integral-valued **streampos** objects. Note that integral-valued **streampos** objects probably are not useful for positioning **fstream** or **stdiostream** objects.

CONSTRUCTORS

This class defines two constructors:

streampos::streampos()
creates a **streampos** object with an unknown value.

streampos::streampos(long n)
creates a **streampos** object from the value **n** . All kinds of stream buffers support the following values of **n** :

streampos(0)
indicates the beginning of the stream.

streampos(EOF)
indicates the end of the stream.

strstream objects created from other values of **n** are not useful for positioning **fstream** or **stdiostream** objects.

MEMBER FUNCTIONS

This class defines two member functions:

streampos::operator long()
reduces the value of the **streampos** to a long integer. The value of **long(streampos(long_val))** is defined to be equal to **long_val** . The result of converting a **streampos** not constructed from a **long** is undefined.

fpos_t* streampos::fpos()
returns a pointer to an **fpos_t** contained in the **streampos** . This **fpos_t** contains a valid value only if this **streampos** was returned from a call to **seekoff()** or **seekpos()** on a **filebuf** or **stdiobuf** object.

class strstream, istrstream, and ostrstream

Provide Formatted String I/O

SYNOPSIS

```
#include <strstream.h>

class istrstream : public strstreambuf,
                   public istream
{
public:
   istrstream(char *str);
   istrstream(char *str, int size);
   ~istrstream( );
   strstreambuf* rdbuf ( );
};

class ostrstream : public strstreambuf,
                   public ostream
{
public:
   ostrstream(char *str, int size,
              int mode = ios::out);
   ostrstream( );
   ~ostrstream( );

   char* str( );
   int pcount ( );
   strstreambuf* rdbuf( );
};

class strstream : public strstreambuf,
                  public iostream
{
public:
   ostrstream(char *str, int size,
              int mode = ios::out);
   ostrstream( );
   ~ostrstream( );

   char* str( );
   int pcount( );
   strstreambuf* rdbuf( );
};

class strstream : public strstreambuf,
                  public iostream
{
public:
   strstream(char *str, int size,
             int mode = ios::out);
   strstream( );
   ~strstream( );

   char* str( );
   int pcount( );
   strstreambuf* rdbuf( );
};
```

DESCRIPTION

class strstream and its associated classes **class istrstream** and **class ostrstream** are declared in the **strstream.h** header file. These classes support string (array) I/O. They do this by customizing the I/O operations defined in the base classes **istream** , **ostream** , and **iostream** .

PARENT CLASSES

class istrstream inherits characteristics from **class istream** . **class ostrstream** inherits characteristics from **class ostream** , and **class strstream** inherits characteristics from **class iostream** . All three of **class istrstream** , **ostrstream** , and **strstream** inherit characteristics from **class ios** . See the descriptions of these parent classes for the details on functions and operators that are inherited.

CONSTRUCTORS

class strstream , **class istrstream** , and **class ostrstream** each have two constructors. For a discussion of dynamic mode versus static mode streams, see "class strstreambuf" on page 91 .

istrstream::istrstream(char *str)
creates a static mode **istrstream** such that extraction operations on the stream will fetch the characters of **str** , up to the terminating ' **\0** '. **str** must be null-terminated. The ' **\0** ' character is not fetched. Seeks are allowed within the array.

istrstream::istrstream(char *str, int size)
creates a static mode **istrstream** such that extraction operations on the stream will fetch characters from the array starting at **str** and extending for **size** bytes. Seeks are allowed within the array.

ostrstream::ostrstream(char *str, int size, int mode = ios::out)
creates a static mode **ostrstream** referencing an area of **size** bytes starting at the character pointed to by **str** . The get pointer is positioned at the beginning of the array. The put pointer is also positioned at the beginning of the array unless either the **ios::ate** or **ios::app** bit is set in **mode** ; if either of these bits is set, the put pointer is positioned at the space that contains the first null character. Seeks are allowed anywhere within the array.

ostrstream::ostrstream()
creates a dynamic mode **ostrstream** . This involves dynamically allocating space to hold stored characters. Seeks are not allowed.

strstream::strstream(char *str, int size, int mode = ios::out)
creates a static mode **strstream** referencing an area of **size** bytes starting at the character pointed to by **str** . The get pointer is positioned at the beginning of the array. The put pointer is also positioned at the beginning of the array unless either the **ios::ate** or **ios::app** bit is set in **mode** ; if either of these bits is set, the put pointer is positioned at the space that contains the first null character.

strstream::strstream()
creates a dynamic mode **strstream** . This involves allocating space to hold stored characters. Seeks are not allowed. The get pointer is positioned at the beginning of the array.

DESTRUCTORS

class istrstream , **class ostrstream** , and **class strstream** each have one destructor:

istrstream::~istrstream() ostrstream::~ostrstream()
strstream::~strstream()
closes the stream. For dynamic stream objects, closing means delete the array, unless it has been frozen. For static stream objects, closing is meaningless.

MEMBER FUNCTIONS

The following functions are members of **class istrstream** , **class ostrstream** , and **class strstream** .

char* ostrstream::str()
char* strstream::str()
call **str()** on the associated **streambuf** . These functions return whatever the **streambuf::str()** call returned.

int ostrstream::pcount()
int strstream::pcount()
return the number of stored bytes.

strstreambuf* istrstream::rdbuf()
strstreambuf* ostrstream::rdbuf()
strstreambuf* strstream::rdbuf()
return a pointer to the **strstreambuf** associated with the stream.

EXAMPLE

This example creates a **strstream** , inserts a string and a number, then extracts them again, writing the contents of **mystream** to **cout** .

```
strstream mystream;
float mynum=3.2;
float num2;

    // Write mynum to mystream.
mystream << mynum << ends;

    // Extract the contents of mystream
    // and store them in num2.
mystream >> num2;

    // Get the string from mystream and
    // write it to cout.
cout << mystream.str();
```

SEE ALSO

```
class strstreambuf
```

Buffer Class Descriptions

This section provides the descriptions of the buffer classes, such as **filebuf** and **strstreambuf** .

Using these classes is an advanced programming technique and is not required for simple C++ programs that use I/O. As with the stream class descriptions, the protected interface to the buffer classes is not included in the class descriptions that follow, although it is implemented.

Each class (or occasionally, a set of related classes) is listed alphabetically. All class descriptions include the following information:

- □ a synopsis of the class
- □ a brief description of the purpose and structure of the class
- □ a discussion of parent classes
- □ a detailed description of the members of the class
- □ examples of using some of the members, if appropriate
- □ a SEE ALSO section that points you to related classes.

In addition, some class descriptions contain other sections, as appropriate for the class.

The following classes are described in this section:

```
class filebuf    class stdiobuf
class streambuf  class strstreambuf
class bsambuf    class bsam_exit_list
```

The **`bsambuf`** class is a specialized version of **`class streambuf`** and was added with Release 6.00. It implements I/O via the record-oriented Basic Sequential Access Method (BSAM) interface of the SAS/C OS Low-Level I/O functions. In addition to providing all the functionality of the **`streambuf`** class, **`class bsambuf`** permits functions to be called as BSAM exits via objects of **`class bsam_exit_list`** .

The **`bsambuf`** class is defined in the header file **`<bsamstr.h>`** . This header file also defines a set of classes for performing formatted file I/O using a **`bsambuf`** object. These include:

- **`class bsamstream`**
- **`class ibsamstream`**
- **`class obsamstream`** .

These classes are specialized versions of class **`iostream`**, **`istream`** , and **`ostream`** , respectively.

For details about the streams library and the parent class of **`class bsambuf`** , refer to "class streambuf" on page 90 in the "I/O Class Descriptions" section. For details about the OS Low-Level I/O functions, refer to *SAS/C Library Reference, Third Edition, Volume 2*. For more information about BSAM, refer to the IBM publication *MVS/XA Data Administration Guide* (GC26-4140).

Note: The classes defined in **`<bsamstr.h>`** can be used in a C++ program that uses the Systems Programming Environment (SPE) library. △

Note: The term *character* in the following class and function descriptions refers to either a **`char`** , a **`signed char`** , or an **`unsigned char`** . △

class bsambuf

Provide File I/O via BSAM

SYNOPSIS

```
class bsambuf : public streambuf
{
public:
   enum error_id { Enone,
                   Estorage,
                   Eabend,
                   Eopen,
                   Eclose,
                   Eget,
                   Eput,
                   Efind,
                   Estow,
                   Eflush,
                   Eseek,
                   Etell
                 };
   bsambuf();
   virtual ~bsambuf();
   int is_open();
   bsambuf *open(const char *filename,
      int mode, const char *keywords = 0,
      int willseek = 0,
      bsam_exit_list *user_exits = 0);
   bsambuf *attach(DCB_t *dcb, int mode,
      int willseek = 0);
   int init_directory();
   int delete_member(const char *name);
   int rename_member(const char *old_name,
      const char *new_name);
   int stow(const char *name,
      char action = 'R',
      int user_data_length = 0,
      const void *user_data = 0,
      int alias = 0, int TT = 0,
      int R = 0, int TTRN = 0);
   int add_member(const char *name,
      int user_data_length = 0,
      const void *user_data = 0,
      int alias = 0, int TT = 0,
      int R = 0, int TTRN = 0);
   int replace_member(const char *name,
      int user_data_length = 0,
      const void *user_data = 0,
      int alias = 0, int TT = 0,
      int R = 0, int TTRN = 0);
   int find(const char *name);
   bsambuf *close();
   char dcbrecfm();
   short dcblrecl();
   short dcbblksize();
   DCB_t *getdcb();
   int error_info(error_id& id);
   void clear_error_info();
   void set_user_data(void *user_data);
   void *get_user_data();
   const char *get_ddname();
   const char *get_member();
   virtual streambuf *setbuf(char *buffer,
      int length);
   virtual streampos seekoff(streamoff
      offset,ios::seek_dir dir, int mode);
   virtual streampos seekpos(streampos
      pos, int mode);
   virtual int sync();
};
```

DESCRIPTION

The **`<bsamstr.h>`** header file defines **`class bsambuf`**. The **`bsambuf`** class is a specialization of **`class streambuf`** that implements I/O via the record-oriented BSAM interface of the OS Low-Level I/O functions. **`bsambuf`** objects are intended for use in C++ programs that use the SPE version of the library, but they may also be useful in other contexts.

In addition to the expected **`streambuf`** functionality, **`class bsambuf`** permits functions to be called as BSAM exits via objects of class **`bsam_exit_list`** .

For more information about the OS Low-Level I/O functions, refer to *SAS/C Library Reference, Third Edition, Volume 2, Release 6.00*. For more information about BSAM, refer to the IBM publication *MVS/XA Data Administration Guide* (GC26-4140).

RESTRICTIONS

- □ A file having the DCB characteristic LRECL=X cannot be connected to a **bsambuf** .
- □ "Short records," that is, records in a fixed-length record data set that are shorter than the record length, are padded on the right. If the file was opened in **ios::binary** mode, the record is padded with null (all bits 0) characters, otherwise the record is padded with blank characters. Padding characters are not removed from records on input.
- □ The number of pushback characters is limited to the size of the buffer, which is either the logical record length of the file or the size specified by **setbuf** , whichever is smaller. In no case can characters be pushed back beyond a record boundary.

PARENT CLASSES

class bsambuf inherits characteristics from **class streambuf** . See the description of this parent class for the details on functions and operators that are inherited.

CONSTRUCTORS

class bsambuf defines one constructor:

bsambuf::bsambuf()
constructs a **bsambuf** object for an unopened file.

DESTRUCTORS

Here is the **class bsambuf** destructor:

virtual bsambuf::~bsambuf()
closes the file, if opened.

TYPES

class bsambuf defines one type, **enum error_id** . This type enumerates the values that may be retrieved by the **error_info** function, described later in this section.

NONVIRTUAL MEMBER FUNCTIONS

The following nonvirtual functions are defined in **class bsambuf** . The virtual functions are described later in this section.

int bsambuf::is_open()
returns a value other than 0 if the **bsambuf** is connected to an open file; returns 0 otherwise.

bsambuf *bsambuf::open
(const char *filename, int mode,
const char *keywords = 0,
int willseek = 0,
bsam_exit_list *user_exits = 0)
opens a file named **filename** and connects the **bsambuf** to it. If the open is successful, **open** returns a pointer to the **bsambuf** . If an error occurs during the open, **open** returns 0. **filename** is a DDname optionally followed by a parenthesized member name. **filename** may be in upper- or lowercase. For example, ' **sysin**' is a valid filename, as is ' **obj(subrtn)**'. **mode** is a combination of one or more **enum open_mode** flags. One or more of **ios::in, ios::out** , or **ios::app** must be set. **keywords** is a string of 0 or more DCB macro keywords. The supported keywords are DSORG, RECFM, LRECL, BLKSIZE, OPTCD, NCP, and BUFNO. The keywords and their values can be in either upper- or lowercase. If several keywords are specified, they can be separated by blanks or commas. For example, the **keyword** string ' **recfm=fb,blksize=6400,lrecl=80**' corresponds to the DCB specification DCB=(RECFM=FB,BLKSIZE=6400,LRECL=80). By default, no keywords are used.

DCB attributes may be specified via the keywords string or by the DD statement or TSO ALLOCATE for the DDname. If any attributes are unspecified for a new file, the following default values are used:

- □ If the record format is not specified, it defaults to variable-length, blocked records (RECFM=VB).
- □ If the record length is not specified, the default is 259 for a variable-length file or 80 for a fixed-length file.
- □ If the block size* is not specified, the default is shown in the following table:

* If the program is running on a version of MVS that supports SMS, a default block size is not assigned unless the file has undefined-length records or is a tape.

Format	*Default Blocksize*
RECFM=F	either the logical record length or the largest multiple of the logical record length less than 6144, whichever is larger
RECFM=V	either the logical record length + 4 or 6144, whichever is larger
RECFM=U	255

If the value of **willseek** is 0, seeking is disabled for this file. Any attempt to use either the **seekoff** and **seekpos** function will result in an error. Specify a value other than 0 for this parameter if you intend to use either **seekoff** or **seekpos** on this file. The default value is 0. **user_exits** is a pointer to a **bsam_exit_list** object describing a number of functions to be called as BSAM exit routines. See "class bsam_exit_list" on page 86 for information about this parameter. By default, no user exits are enabled.

Restrictions

- □ The **ios::app** mode flag is treated as **ios::ate**.
- □ The **ios::nocreate** and **ios::noreplace** mode flags are ignored.

bsambuf *bsambuf::attach(DCB_t *dcb, int mode, int willseek = 0)
associates a DCB that has already been created by either the **osdcb** or **osbdcb** function to a **bsambuf** and opens the file. If the open is successful, **attach** returns a pointer to the **bsambuf** . If an error occurs during the open, **attach** returns 0. **mode** and **willseek** have the same meaning as they have in the **open** function.

Restrictions

- □ The DCB cannot already be open.
- □ The **bsambuf** implementation reserves the DCBUSER field for its own use.
- □ The **ios::app** mode flag is treated as **ios::ate**.
- □ The **ios::nocreate** and **ios::noreplace** mode flags are ignored.

int bsambuf::init_directory()
invokes the "initialize directory" function of the STOW macro on the file (which must be a PDS) connected to the **bsambuf** . The value returned is 0 if the function succeeds or is a value other than 0 otherwise.

int bsambuf::delete_member (const char *name)
deletes the PDS member identified by **name** from the file connected to the **bsambuf** . **name** may be specified in either upper- or lowercase. If **name** is shorter than eight characters it will be padded on the right with blanks. The value returned is 0 if the member is successfully deleted or is a value other than 0 otherwise.

int bsambuf::rename_member (const char *old_name, const char *new_name)
renames the PDS member identified by **old_name** to **new_name** . **old_name** and **new_name** may be specified in either upper- or lowercase. If either name is shorter than eight characters it will be padded on the right with blanks. The value returned is 0 if the member is successfully renamed or is a value other than 0 otherwise.

int bsambuf::stow(const char *name, char action = 'R', int user_data_length = 0, const void *user_data = 0, int alias = 0, int TT = 0, int R = 0, int TTRN = 0)
adds or replaces a member or alias in the file connected to the **bsambuf** . The file must be a PDS. **name** is the name of the member or alias. **name** may be specified in either upper- or lowercase. If **name** is shorter than eight characters, it will be padded on the right with blanks. **action** may be either 'R', indicating that **name** replaces an existing member or alias in the PDS, or 'A', which indicates that **name** is to be added to the PDS. The default value is 'R'. **user_data_length** is the length, in bytes, of the user data to be associated with name in the PDS directory. This value must be nonnegative and no greater than 62. **user_data** is a pointer to the user data. The default value of both **user_data_length** and **user_data** is 0.

If **alias** is a value other than 0, then **name** is treated as an alias to be added or replaced. The default value of **alias** is 0. **TT** and **R** are the relative track and record number, respectively, of **name** . These values are ignored when **alias** is 0. By default both are 0.[*] **TTRN** is the number of TTRN fields in **user_data** . This must be a nonnegative number no greater than 3. By default it is 0. **stow** returns a value other than 0 if any of the arguments are out of bounds. If

[*] **class bsambuf** defines no function that can retrieve this information. Use the **osnote** function in the OS Low-Level I/O group of functions to get the TTRz value for the current position in the file.

stow invokes the STOW macro, with one exception, **stow( )** returns the return code from the STOW macro. (This code can also be retrieved via the **error_info** function.) In general, the STOW macro returns 0 if the requested action succeeded, or a value other than 0 otherwise. The exception occurs when **action** is 'R' and the STOW macro returns 8. This return code from the STOW macro indicates that the member or alias did not previously exist and so was added. In this case, **stow** returns 0.

int bsambuf::add_member(const char *name,
int user_data_length = 0,
const void *user_data = 0,
int alias = 0, int TT = 0,
int R = 0, int TTRN = 0)

is equivalent to the **stow** function when **action** is 'A'. The arguments and return value are identical to **stow** .

int bsambuf::replace_member
(const char *name,
int user_data_length = 0,
const void *user_data = 0,
int alias = 0, int TT = 0,
int R = 0, int TTRN = 0)

is equivalent to the **stow** function when **action** is 'R'. The arguments and return value are identical to **stow** .

int bsambuf::find(const char *name)

positions the file, which must be a PDS, to the start of the member identified by **name** . **name** may be specified in either upper- or lowercase. If name is shorter than eight characters it will be padded on the right with blanks. **find** returns 0 if the file is successfully positioned, or a value other than 0 otherwise.

bsambuf *bsambuf::close()

causes any outstanding output to be flushed, then closes the file and disconnects the **bsambuf** from it (even if errors occur). The **bsambuf** 's I/O state is cleared. If the close is successful, **close** returns a pointer to the **bsambuf** . If an error occurs, **close** returns 0.

char bsambuf::dcbrecfm()

returns the value of the DCBRECFM field in the DCB. The bits in this field describe the record format of the file connected to the **bsambuf** . The **<osio.h>** header file contains preprocessor symbol definitions for the bits that may be set in this field. The most commonly used flags are shown in the following table:

Symbol	Format
DCBRECF	fixed
DCBRECV	variable
DCBRECU	undefined (see Note)
DCBRECBR	blocked

Note: DCBRECU is equivalent to ORing DCBRECF with DCBRECV. When examining the value returned by **dcbrecfm** , you must always check for DCBRECU before testing the DCBRECF or DCBRECV flags. △

short bsambuf::dcblrecl()
short bsambuf::dcbblksize()

return the logical record length (LRECL) and block size (BLKSIZE) of the file connected to the **bsambuf** .

DCB_t *bsambuf::getdcb()

returns a pointer to the DCB associated with the file connected to the **bsambuf** .

int bsambuf::error_info(error_id& id)

returns the return code from the first failed low-level routine. If a function receives a value other than 0 as a return code from a low-level routine, it saves the return code and a value of type **error_id** indicating which routine failed. **error_info** retrieves this information, which may be used for a detailed error analysis.

The value returned in the **id** argument may have one of the values shown in the following table. Unless otherwise noted, the failing routine is a BSAM routine in the OS Low-Level I/O group of functions.

error_id	Routine
Enone	(no error)
Estorage	GETMAIN[1]
Eabend	BSAM[2]
Eopen	osopen
Eclose	osclose
Eget	osget[3]
Eput	osput
Efind	osfind
Estow	osstow
Eflush	osflush
Eseek	osseek
Etell	ostell

1 This value is set when storage cannot be allocated for a buffer. The code returned by **error_info** is always 4.

2 This value is returned when an "ignorable" ABEND has occurred. An ignorable ABEND does not terminate the program but does prevent any further I/O from being performed on the file connected to the **bsambuf**. The value

of the return code is that of the completion code field in the ABEND exit parameter list.

3 Although end-of-file causes **osget** to return a value other than zero as a return code, this value is not stored as an error code.

If, after a low-level routine has failed and its error information stored, a subsequent low-level routine fails, the error information for that failure is not stored unless **clear_error_info** has been called.

void bsambuf::clear_error_info()
sets the stored error information to the initial state. In this state, the value of **error_id** is **Enone** and the return code is 0.

void bsambuf::set_user_data
(void *user_data)
stores the value in **user_data** for later retrieval by **get_user_data** . The value may be any value required by the program. It is ignored by the stream.

void *bsambuf::get_user_data()
retrieves the value stored by **set_user_data** .

void *bsambuf::get_ddname()
returns a pointer to the DDname part of the filename argument to **open** . The DDname is uppercased and terminated by a ' **\0** '. If the DCB was associated to the **bsambuf** by the **attach** function, the pointer points to a 0-length string.

void *bsambuf::get_member()
returns a pointer to the member part of the filename argument to **open** . The member name is uppercased and terminated by a ' **\0** '. If the DCB was associated to the bsambuf by the **attach** function, or the filename did not specify a member name, the pointer points to a 0-length string.

VIRTUAL MEMBER FUNCTIONS

virtual streambuf *bsambuf::setbuf
(char *buffer, int length)
offers the character array starting at **buffer** and containing **length** bytes as a buffer for use by the **bsambuf** . If **buffer** is 0 or length is less than or equal to 0, the **bsambuf** is unbuffered. (However, buffering by BSAM will still take place.) This function must be called before any I/O is requested for this **bsambuf** and can be called only once for the **bsambuf** . If the buffer will be used by the **bsambuf** , **setbuf** returns a pointer to the **bsambuf** .

If this function is called after I/O has been requested for the **bsambuf** , it does nothing and returns 0. If this function is called more than once, subsequent calls for the **bsambuf** do nothing except return 0. This function does not affect the I/O state of the **bsambuf** .

By default, the buffer size is equal to the number of data characters in a record. In most cases, use of the **setbuf** function will have little, if any, effect on I/O performance. However, changing the **bsambuf** to be unbuffered will severely degrade the performance. Also note that the underlying BSAM routines will always buffer the data, whether the **bsambuf** is unbuffered or buffered.

virtual streampos bsambuf::seekoff
(streamoff offset, ios::seek_dir dir, int mode)
sets the get and put pointers to a new position, as indicated by **offset** and **dir** . For a **bsambuf** , **seekoff** will only accept 0 as the value of **offset**. The only acceptable values for **dir** are **ios::beg** and **ios::cur** . (Refer to the description of **streambuf::seekoff** for an explanation of these two values.) If **dir** is **ios::beg** , the file is positioned to the beginning. If **dir** is **ios::cur**, the position of the file is not changed. The new location of the file is returned in the **streampos** .

For **bsambuf** objects, the get and put pointers are the same, so the **mode** argument is ignored. **seekoff** returns **(streampos) EOF** if **offset** is a value other than 0, if **dir** is neither **ios::beg** or **ios::cur** , or if the file position cannot be moved. **seekoff** will also always return **(streampos) EOF** if **willseek** was 0 when the file was connected to the **bsambuf** by either **open** or **attach** .

virtual streampos bsambuf::seekpos
(streampos pos, int mode)
sets the get and/or put pointers to a new position, as indicated by **streampos** . This function returns the new position, or **seekpos(EOF)** if an error occurs. For **bsambuf** objects, the get and put pointers are the same, so the **mode** argument is ignored.

virtual int sync()
tries to force the state of the get and put pointers of the **bsambuf** to be synchronized with the state of the file it is associated with. If some characters have been buffered for output they will be written to the file.* Characters that have been buffered for input will be discarded. Note that synchronization is not possible unless the file is positioned at a record boundary.

This function returns 0 if synchronization succeeds, otherwise it returns EOF.

SEE ALSO
class bsam_exit_list, class bsamstream,
class ibsamstream, class obsamstream

* The **sync** function invokes the low-level **osflush** routine to flush all buffered characters to the operating system. Note, however, that all characters may not be written immediately, due to additional buffering performed by the operating system.

class bsam_exit_list

Define BSAM Exit Routines

SYNOPSIS

```
#include <bsamstr.h>

class bsam_exit_list {
   bsam_exit_list();
   bsam_exit_list(exit_t *list);
   int use(_e_exit code, _e_exit_fp exit);
   int use(_e_exit code, void *exit);
   int remove(_e_exit code,_e_exit_fp exit);
   int remove(_e_exit code, void *exit);
   };
```

DESCRIPTION

The **<bsamstr.h>** header file defines **class bsam_exit_list** . A pointer to a **bsam_exit_list** object can be passed to the **bsambuf::open** , **bsamstream::open** , **ibsamstream::open** , or **obsamstream:open** function to define one or more functions that are to be used as DCB exit routines.

RESTRICTIONS

- □ Functions that serve as DCB exit routines cannot be member functions.
- □ Only one function can be called for each type of DCB exit.
- □ No more than 18 exits can be specified in each **bsam_exit_list** object. OS Low-Level I/O defines only 18 distinct exits.

CONSTRUCTORS

class **bsam_exit_list** defines two constructors:

bsam_exit_list::bsam_exit_list()
constructs an object that defines no exits.

bsam_exit_list::bsam_exit_list (exit_t list[])
constructs an object that defines the exits specified by **list** . **list** is an array of type **exit_t** . This type is defined as shown here:

```
typedef struct {
         unsigned exit_code;
         union {
           _e_exit_fp exit_addr;
           void *area_addr;
          };
     } exit_t;
```

exit_code may be any one of the values defined for the enumeration type **_e_exit** , which is also defined as shown here:

```
enum _e_exit {INACTIVE = 0, INHDR = 1,
    OUTHDR = 2, INTLR = 3, OUTTLR = 4,
    OPEN = 5, EOV = 6, JFCB = 7,
    USER_TOTAL = 10, BLK_COUNT = 11,
    DEFER_INTLR = 12,
    DEFER_NONSTD_INTLR = 13, FCB = 16,
    ABEND = 17, JFCBE = 21,
    TAPE_MOUNT = 23, SECURITY = 24,
    LAST = 128, SYNAD = 256};
```

The last entry in the list must set **area_addr** to 0 and have an **exit_code** value of LAST or, if either **exit_addr** or **area_addr** is a value other than 0, you must OR the **exit_code** value with LAST.

Here is an example using this constructor:

```
static int open_exit(void *r1, void *r0)
   {
   // DCB open exit processing...
   return 0;
   }
static int abend_exit(void *r1, void *r0)
   {
   // DCB abend exit processing...
   return 0;
   }
int main()
 {
 exit_t exit_array[] = {{OPEN, open_exit},
                     {ABEND, abend_exit},
                     {LAST, 0}
                      } ;
   // Construct "list" with
   // two DCB exits.
     bsam_exit_list list(exit_array);

   // Open DDname "OUT"
   // defining two DCB exits.
     obsamstream o("out", ios::out, 0, 0,
                  &list);

   /* remainder of program */
     .
     .
     .
```

DESTRUCTORS

Here is the **class bsam_exit_list** destructor:

virtual bsam_exit_list::~bsam_exit_list()

NONVIRTUAL MEMBER FUNCTIONS

The following nonvirtual functions are defined in **class bsam_exit_list** . This class defines no virtual functions.

Note: BSAM exits may be either functions or data areas. Therefore, class **bsam_exit_list** defines functions for both types. △

int use(_e_exit code, _e_exit_fp exit)
adds a pointer to a function to be used as a DCB exit to the **bsam_exit_list** . The pointer must have type **_e_exit_fp** , which is defined as:

```
typedef __remote int (*_e_exit_fp)
       (void *, void *)
```

The exit must be a static member function or a nonmember function. The BSAM exit type is specified by **code** . If the exit is added, **use** returns 0. Otherwise it returns a value other than 0.

On entry to a BSAM exit, the first argument is the value of general register 1 as established by BSAM. Likewise, the second argument is the value of general register 0.

ABEND exit considerations

A default DCB ABEND exit is always defined for a **bsambuf** . If you define an DCB ABEND exit for the **bsambuf** , your exit will be run first, followed by the default exit. Your exit must set the option mask byte in the ABEND exit parameter list before returning. If the option mask byte is set to 4, indicating that the ABEND should be "ignored," the default exit causes the DCB to be closed and freed.

For your convenience, the **<bsamstr.h>** header file contains a partial definition of the parameter list BSAM creates for the DCB ABEND exit. On entry to the exit function, general register 1 (the first argument) contains a pointer to this parameter list:

```
struct Abend_exit_parms {
        char completion_code :12;
        char                 :4;
        char return_code;
        union {
           struct {
              char         :4;
              char recover :1;
              char ignore  :1;
              char delay   :1;
              char         :1;
              } can;
           char option_mask;
           };
        DCB_t *dcb;
       };
     // Set "option_mask" to this value
     // to ignore the ABEND.
     const int IGNORE_ABEND = 4;
```

int remove(_e_exit code, _e_exit_fp exit)

removes an exit function from the **bsam_exit_list** . If an exit having the matching exit code and function pointer is not in the **bsam_exit_list** , **remove** returns a value other than 0. Otherwise, **remove** removes the exit and returns 0.

int use(_e_exit code, void *exit)

adds a data pointer to be used as a DCB exit to the **bsam_exit_list** . The BSAM exit type is specified by **code** .

If the exit is added, **use** returns 0. Otherwise it returns a value other than 0.

int remove(_e_exit code, void *exit)

removes a data exit from the **bsam_exit_list** . If an exit having the matching exit code and address is not in the **bsam_exit_list** , **remove** returns a value other than 0. Otherwise, **remove** removes the exit and returns 0.

SEE ALSO

class bsambuf, class bsamstream,
class ibsamstream, class obsamstream

class filebuf

Provide File I/O

SYNOPSIS

```
#include <fstream.h>
class filebuf : public streambuf
{
public:
   filebuf();
   virtual ~filebuf();
   int is_open();
   filebuf* open(const char *name,
                 int mode,
                 const char *amparms = "",
                 const char *am = "");
   filebuf* close();
   virtual streampos seekoff(streamoff
         offset, seek_dir place,
         int mode = ios::in|ios::out);
   virtual streampos seekpos(streampos pos,
         int mode = ios::in|ios::out);
   virtual streambuf* setbuf(char *p,
                             size_t len);
   virtual int sync();
};
```

DESCRIPTION

The **fstream.h** header file defines **class filebuf** . **filebuf** objects represent the lowest level of file I/O that is standard C++. They provide a specialized form of **streambufs** that uses a file as the source or destination (sink) for characters. Input corresponds to file reads and output corresponds to file writes. For **filebuf** objects, the get and put pointers are tied together. That is, if you move one, you move the other. If the file has a format that allows seeks, a **filebuf** allows seeks. **filebuf** I/O guarantees at least four characters of putback. You do not need to perform any special action between reads and writes (in contrast to standard C I/O, where such seeks are required).

When a **filebuf** is connected to a file, the **filebuf** is said to be open. There is no default open mode, so you must always specify the open mode when you create a **filebuf** .

PARENT CLASSES

class filebuf inherits characteristics from **class streambuf** . See the description of this parent class for the details on functions and operators that are inherited.

CONSTRUCTORS

class filebuf defines one constructor:

filebuf::filebuf()
creates an unopened file.

DESTRUCTORS

Here is the **class filebuf** destructor:

virtual filebuf::~filebuf()
closes the file, if opened.

NONVIRTUAL MEMBER FUNCTIONS

The following nonvirtual functions are defined in **class filebuf** . The virtual functions are described later in this section.

int filebuf::is_open()
returns a nonzero value if the **filebuf** is connected to an open file; returns 0 otherwise.

filebuf* filebuf::open(const char *name,
int mode, const char *amparms = "",
const char *am = "")
opens a file named **name** and connects the **filebuf** to it. If the open is successful, **open()** returns a pointer to the **filebuf** . If an error occurs during the open (for example, if the file is already open), **open()** returns 0. See "enum open_mode" on page 70 for a description of the **mode** argument.

An explanation of filename specification and the arguments **amparms** and **am** can be found in the *SAS/C Library Reference, Volume 1.* (Note that the order of the **amparms** and **am** arguments in this function is the opposite of the order in which they appear in calls to the C **afopen** function.) You may also want to refer to the *SAS/C Compiler and Library User's Guide*.

filebuf* filebuf::close()
causes any outstanding output to be flushed, then closes the file and disconnects the **filebuf** from it (even if errors occur). Also, the **filebuf** 's I/O state is cleared. If the close is successful, **close()** returns a pointer to the **filebuf** . If an error occurs during the close, **close()** returns 0.

VIRTUAL MEMBER FUNCTIONS

The following functions override their base class definitions (in **class streambuf**).

virtual streampos filebuf::seekoff
(streamoff offset, seek_dir place,
int mode = ios::in|ios::out)
sets the get and put pointers to a new position, as indicated by **place** and **offset** . (Descriptions of **offset** and **place** are contained in the **streambuf::seekoff()** description.) This function returns the new position, or it returns **streampos(EOF)** if an error occurs (for example, the file may not support seeking, or you may have requested a seek to a position preceding the beginning of the file). The position of the file after an error is undefined. Some files support seeking in full, and some impose lesser or greater restrictions on seeking. **seekoff()** corresponds to the C **fseek** function, and **seekpos()** corresponds to the C **fsetpos** function. Rules for these similar C functions are given in the *SAS/C Library Reference, Volume 1, Third Edition, Release 6.00*.

For **filebuf** objects, the get and put pointers are the same (moving either one moves the other). Because of this, you do not have to use the last argument, **mode** .

virtual streampos filebuf::seekpos
(streampos pos,
int mode = ios::in|ios::out)
sets the get and/or put pointers to a new position, as indicated by **streampos** . This function returns the new position, or it returns **seekpos(EOF)** if an error occurs. For **filebuf** objects, the get and put pointers are the same (moving either one moves the other). Because of this, you do not have to use the last argument, **mode** .

virtual streambuf* filebuf::setbuf
(char *p, size_t len)
offers the character array starting at **p** and containing **len** bytes as a buffer for use by the **filebuf** . If **p** is null or **len** is less than or equal to 0, the **filebuf** is unbuffered. (However, buffering by the SAS/C Library and the operating system may still take place.) This function must be called before any I/O is requested for this **filebuf** and can be called only once for the **filebuf** . Under normal conditions, **setbuf()** returns a pointer to the **filebuf** .

If this function is called after I/O has been requested for the **filebuf** , this function does nothing and returns **NULL** . If this function is called more than once, subsequent calls for the **filebuf** do nothing except return **NULL** . This function does not affect the I/O state of the **filebuf** .

virtual int filebuf::sync()
tries to force the state of the get or put pointer of the **filebuf** to be synchronized with the state of the file it is associated with. This attempt at synchronization may result in the following:

- characters being written to the file, if some have been buffered for output. Note that all characters may not be written immediately due to additional buffering performed by the operating system.
- an attempt to seek the file, if characters have been read and buffered for input.

This function usually returns 0; if synchronization is not possible, it returns EOF.

IMPLEMENTATION

Usually, **filebuf** objects directly access the native I/O facilities of the system on which they are implemented. For this release of the SAS/C C++ Development System, **filebuf** objects are implemented in terms of C **FILE*** s. This may be changed in later versions of this library, and no assumptions should be made of this particular implementation.

SEE ALSO

```
class fstream, class ifstream,
class ofstream
```

class stdiobuf

Provide I/O in a Mixed C and C++ Environment

SYNOPSIS

```
#include <stdiostream.h>

class stdiobuf : public streambuf
{
public:
  stdiobuf(FILE *file);

  virtual ~stdiobuf( );

  int is_open( );

  FILE* stdiofile( );

  streampos seekoff(streamoff offset,
                    seek_dir place,
                    int mode =
                    ios::in|ios::out);
  streampos seekpos(streampos pos,
                    int mode =
                    ios::in|ios::out);
  virtual int sync();
};
```

DESCRIPTION

The **stdiostream.h** header file declares **class stdiobuf** . **stdiobufs** are intended to be an interface to ANSI C style **FILE** *s on those systems that provide **FILE** *s. Calls to **stdiobuf** member functions are mapped directly to calls to ANSI C **stdio** functions.

Because **stdiobuf** objects provide no buffering other than that provided by the C **stdio** functions, any changes to file attributes or contents made via a **stdiobuf** are reflected immediately in the **stdio** data structures. This includes file positioning using **seekoff()** or **seekpos()** . For **stdiobuf** objects, the get and put pointers are tied together. That is, if you move one, you move the other.

Unless you are mixing **streambuf** and **stdio** access to the same file, you should use **class filebuf** instead of **class stdiobuf** . Use of **filebuf** objects may improve performance.

PARENT CLASSES

class stdiobuf inherits characteristics from **class streambuf** . See the description of this parent class for the details on functions and operators that are inherited.

CONSTRUCTORS

class stdiobuf defines one constructor:

stdiobuf::stdiobuf(FILE *file)
creates a **stdiobuf object** associated with an open **FILE** *.

DESTRUCTORS

Here is the **class stdiobuf** destructor:

virtual stdiobuf::~stdiobuf()
closes the associated **FILE** *, if opened.

NONVIRTUAL MEMBER FUNCTIONS

The following descriptions detail the nonvirtual member functions for **class stdiobuf** . The redefined virtual functions are described later in this section.

int stdiobuf::is_open()
returns a nonzero value if the **stdiobuf** is connected to an open file; returns 0 otherwise.

FILE* stdiofile()
returns the associated **FILE** *.

VIRTUAL MEMBER FUNCTIONS

The following functions override their base class definitions (in **class streambuf**).

streampos stdiobuf::seekoff
(streamoff offset, seek_dir place,
int mode = ios::in|ios::out)
moves the get and/or put pointers of the **streambuf** . **place** can be one of the following:

ios::beg
indicates the start of file.

ios::cur
indicates the current get and put position.

ios::end
indicates the end of file.

offset is a positive or negative integer position relative to place. **mode** can be one of the following:

ios::in
moves the get pointer.

ios::out
moves the put pointer.

ios::in|ios::out
moves both pointers.

Whether **seekoff()** and **seekpos()** work for an **stdiobuf** depends on the characteristics of the associated **FILE*** . See the *SAS/C Library Reference, Volume 1, Third Edition, Release 6.00* for more information on **FILE*** characteristics.

streampos stdiobuf::seekpos
(streampos pos,
int mode = ios::in|ios::out)
moves the get and/or put pointers of the **streambuf** . **pos** must be a value returned by a previous call to **seekoff()** . **mode** can be one of the following:

ios::in
moves the get pointer.

ios::out
moves the put pointer.

ios::in|ios::out
moves both pointers.

Some stream buffers do not support seeking. For those stream buffers, **seekpos()** returns **streampos(EOF)** to indicate an error occurred. See the documentation for particular stream buffer classes (such as **filebuf**) for more information on what kinds of seeking are allowed.

virtual int stdiobuf::sync()
tries to force the state of the get or put pointer of the **stdiobuf** to be synchronized with the state of the associated file. This attempt at synchronization may result in the following:

- buffered for output. Note that all characters may not be written immediately due to additional buffering performed by the operating system.
- an attempt to seek the file, if characters have been read and buffered for input.

This function usually returns 0; if synchronization is not possible it returns EOF.

SEE ALSO

```
class stdiostream
```

class streambuf

Provide Base Class for All Stream Buffers

SYNOPSIS

```
#include <iostream.h>

class streambuf
{
public:
   int in_avail();
   int out_waiting();

   int sbumpc();
   int sgetc();
   int sgetn(char *s, int n);
   int snextc();
   void stossc();

   int sputbackc(char c);
   int sputc(int c);
   int sputn(const char *s, int n);

   virtual int sync();
   virtual streampos seekoff(streamoff
            offset, seek_dir place,
            int mode = ios::in|ios::out);
   virtual streampos seekpos(streampos pos,
            int mode = ios::in|ios::out);
   virtual streambuf* setbuf(char *p,
                             int len);
};
```

DESCRIPTION

class streambuf is declared in the **iostream.h** header file and is the base class for all stream buffers. Stream buffers manage the flow of characters between the program and the ultimate sources or consumers of characters, such as external files. The **streambuf** class defines behavior common to all stream buffers. More specialized classes can be derived from **class streambuf** to implement appropriate buffering strategies for particular stream types. For instance, **filebufs** implement buffering suitable for file input or output and **strstreambufs** implement buffering suitable for transfer of data from strings in memory. A **streambuf** is almost never used directly (classes derived from it are used instead) but more often acts as an interface specification for derived classes.

The functions defined by the **streambuf** interface are divided into two groups: nonvirtual functions and virtual functions. These sets of functions are described separately.

CONSTRUCTORS AND DESTRUCTORS

class streambuf defines two constructors and one destructor. All these functions are protected. This ensures that **class streambuf** is used only as a base class for derived classes.

NONVIRTUAL MEMBER FUNCTIONS

The following list describes the nonvirtual **streambuf** interface.

int streambuf::in_avail()
returns the number of characters that have been buffered for input, that is, the number of characters that have been read from the ultimate source of the input but have not been extracted from the **streambuf** . Generally, this information is useful only for classes derived from **class streambuf** .

int streambuf::out_waiting()
returns the number of characters that have been buffered for output, that is, the number of characters that have been inserted into the **streambuf** but have not been delivered to its ultimate destination. Generally, this information is useful only for classes derived from **class streambuf** .

int streambuf::sbumpc()
advances the get pointer one character and returns the character preceding the advanced pointer. If the get pointer is at the end of the stream, the get pointer is not moved and EOF is returned.

int streambuf::sgetc()
returns the character following the get pointer. This function does not move the get pointer. If the get pointer is at the end of the stream, this function returns EOF.

int streambuf::sgetn(char *s, int n)
extracts the next n characters from the stream into **s** and positions the get pointer after the last extracted character. If there are less than

n characters between the get pointer and the end of the stream, those characters are extracted into **s** and the get pointer is moved to the end of the stream. This function returns the number of characters extracted into **s** .

int streambuf::snextc()
advances the get pointer one character and returns the character after the advanced pointer. If the get pointer is at the end of the stream, the get pointer is not moved and EOF is returned.

void streambuf::stossc()
advances the get pointer one character.

int streambuf::sputbackc(char c)
backs the get pointer up one character, returning **c** . **c** must be the character that the get pointer is moved over; if it is not, the effects of this function are undefined. For example, if the get pointer is at the start of the stream and you call this function, the effect is undefined.

Also, each class derived from **streambuf** may impose a limit on the number of characters that can be moved over by **sputbackc()** . If you exceed this limit, this function returns EOF. For some classes, you cannot back up over any characters.

int streambuf::sputc(int c)
stores **c** in the position following the put pointer, replacing any pre-existing character, and then advances the put pointer one position. This function returns **c** if the operation is successful, or EOF if an error occurs.

int streambuf::sputn(const char *s, int n)
stores after the put pointer the first **n** characters addressed by **s** , replacing any pre-existing characters in those positions, and then advances the put pointer past the last character stored. This function returns the number of characters successfully stored.

VIRTUAL MEMBER FUNCTIONS

The following virtual functions are members of **class streambuf** . These functions can be redefined in derived classes (both those supplied by the library and those you define yourself) to customize the behavior of **streambuf** objects.

virtual int streambuf::sync()
tries to force the state of the get or put pointer of the **streambuf** to be synchronized with the state of the sink or source it is associated with. This function returns 0 if successful, or EOF if an error occurs.

virtual streampos streambuf::seekoff (streamoff offset, seek_dir place, int mode = ios::in|ios::out)
moves the get and/or put pointers of the **streambuf** . **place** can be one of the following:

ios::beg
indicates the start of file.

ios::cur
indicates the current get or put position.

ios::end
indicates the end of file.

offset is a positive or negative integer position relative to place. **mode** can be one of the following:

ios::in
moves the get pointer.

ios::out
moves the put pointer.

ios::in|ios::out
moves both pointers. This is the default value.

Some kinds of seeking are not supported for certain stream buffers. If a particular stream buffer does not support the seeking specified, this function returns **streampos(EOF)** to indicate an error occurred. See the documentation for particular stream buffer classes (such as **filebuf**) for more information on what kinds of seeking are allowed.

virtual streampos streambuf::seekpos (streampos pos, int mode = ios::in|ios::out)
moves the get and/or put pointers of the **streambuf** . **pos** must be a value returned by a previous call to **seekoff()** . **mode** can be one of the following:

ios::in
moves the get pointer.

ios::out
moves the put pointer.

ios::in|ios::out
moves both pointers. This is the default value.

Some stream buffers do not support seeking. For those stream buffers, **seekpos()** returns **streampos(EOF)** to indicate an error occurred. See the documentation for particular stream buffer classes (such as **filebuf**) for more information on what kinds of seeking are allowed.

virtual streambuf* streambuf::setbuf (char *p, int len)
allocates a buffer area to be used for buffering within the **streambuf** .

class strstreambuf

Provide String I/O

SYNOPSIS

```
#include <strstream.h>

class strstreambuf : public streambuf
```

```
{
public:
   strstreambuf( );
   strstreambuf(int len);
   strstreambuf(void* (*a) (long),
                void (*f) (void*));
   strstreambuf(char *b, int size,
                char *pstart = 0);

   ~strstreambuf( );

   void freeze(int n = 1);
   char* str( );

   virtual streambuf* setbuf(char *p,
                             int len);
   inc sync( );
   virtual streampos seekoff
           (streamoff offset,
            seek_dir place,
            int mode = ios::in|ios::out);

   virtual streampos seekpos
            (streampos pos,
            int mode = ios::in|ios::out);
};
```

DESCRIPTION

The header file **strstream.h** declares **class strstreambuf** , which specializes **class streambuf** to provide for I/O using arrays of **char** (strings).

For **strstreambuf** objects, the get and put pointers are separate. That is, if you move one of these pointers you do not necessarily move the other. **strstreambuf** objects are created in one of two different modes, dynamic mode or static (fixed) mode. Once a **strstreambuf** is created, it does not change modes. The following list explains the difference between fixed and dynamic mode:

dynamic mode
: means the **strstreambuf** does not have a fixed size and grows as needed. When the array associated with a dynamic mode **strstreambuf** is filled, the **strstreambuf** automatically allocates a larger array, copies the old smaller array to the larger, and frees the smaller array. The functions used to handle allocating and freeing the arrays are determined by the constructor used to create the **strstreambuf** (see the description of constructors for "class strstreambuf" on page 91 .

static mode
: means the **strstreambuf** has a fixed size that does not change. If the array associated with a static mode **strstreambuf** is filled, further writes to the **strstreambuf** may corrupt memory. Be cautious when inserting into static mode **strstreambuf** s.

Note: Do not confuse static mode with the **static** storage class modifier. △

class strstreambuf defines some member functions of its own and also redefines several virtual functions from the base class.

PARENT CLASSES

class strstreambuf inherits characteristics from **class streambuf** . See the description of this parent class for the details on functions and operators that are inherited.

CONSTRUCTORS

class strstreambuf defines four constructors:

strstreambuf::strstreambuf()
creates an empty strstreambuf object in dynamic mode.

strstreambuf::strstreambuf(int len)
creates an empty strstreambuf object in dynamic mode. The initial allocation uses at least len bytes.

strstreambuf::strstreambuf
(void*(*a)(long), void (*f)(void*))
creates an empty **strstreambuf** object in dynamic mode. **a** is the allocation function to be used to do the dynamic allocation and takes as its argument a **long** , which specifies the number of bytes to allocate. If **a** is **NULL** , the **operator new** is used instead of **a** . **f** is the deallocation function, which frees the space allocated by **a** . **f** takes as its argument a pointer to an array allocated by **a** . If **f** is **NULL** , the **operator delete** is used instead of **f** .

strstreambuf::strstreambuf(char *b,
int size, char *pstart = 0)
constructs a **strstreambuf** object in static mode; it does not grow dynamically. **b** specifies where to start the array and **size** specifies the size of the array, as explained in the following list.

- □ If **size** is positive, the array is **size** bytes long.
- □ If **size** is 0, the function assumes **b** points to the start of a null-terminated string. In this case, the string, not including the ' **\0** ' character, is considered to be the **strstreambuf** .
- □ If **size** is negative, the **strstreambuf** is assumed to be indefinitely long.

The get pointer receives the value of **b** and the put pointer receives the value of **pstart** . If **pstart** is **NULL** , then storing characters in the **strstreambuf** is not allowed and causes the function to return an error.

DESTRUCTORS

Here is the **class strstreambuf** destructor:

strstreambuf::~strstreambuf()
closes the **strstreambuf** object. The destructor causes any memory allocated for the **strstreambuf** to be freed.

NONVIRTUAL MEMBER FUNCTIONS

class strstreambuf defines two nonvirtual member functions.

void strstreambuf::freeze(int n = 1)
controls the automatic deletion of the array. If **n** is nonzero, which is the default, the array is not deleted automatically. If n is 0, the array is unfrozen and is deleted automatically. The array is deleted whenever a dynamically created **strstreambuf** needs more space or when the destructor is called. This function only applies to dynamically created **strstreambuf** s; it has no effect on statically created **strstreambuf** s.

If you try to store characters in a frozen array, the effect is undefined.

char* strstreambuf::str()
returns a pointer to the first character in the current array and freezes the array. After **str()** has been called, the effect of storing characters in the array is undefined until the strstreambuf is unfrozen by calling **freeze(0)** .

VIRTUAL MEMBER FUNCTIONS

class strstreambuf redefines several virtual functions from its base class (**class streambuf**).

virtual strstreambuf::streambuf* setbuf (char *p, int len)
tells the **strstreambuf** that the next time an array is dynamically allocated it should be at least **len** bytes long. **p** is ignored.

int strstreambuf::sync()
returns EOF.

virtual streampos strstreambuf::seekoff (streamoff offset, seek_dir place, int mode = ios::in|ios::out)
moves the get and/or put pointers of the **strstreambuf** . See the description of **streambuf::seekoff** for explanations of **offset** , **place** and **mode** .

If the **strstreambuf** is in dynamic mode, this function returns **streampos(EOF)** to indicate an error occurred.

If either the get or put pointer is moved to a position outside the **strstreambuf** , or if the put pointer is moved for a **strstreambuf** that does not allow output, then **streampos(EOF)** is returned and the pointers are not moved.

If **place** is **ios::end** , it refers to the end of the array.

virtual streampos strstreambuf::seekpos (streampos pos, int mode = *ios::in|ios::out)
moves the get and/or put pointers of the **strstreambuf** .

If the **strstreambuf** is in dynamic mode, this function returns **streampos(EOF)** to indicate an error occurred. **pos** must be a value returned by a previous call to **seekoff()** .

See the description of **streambuf::seekpos()** for an explanation of **mode** . If **ios::out** is specified for **mode** and output is not allowed for this **strstreambuf** , then **streampos(EOF)** is returned to indicate an error occurred and the put pointer is not moved.

SEE ALSO

```
class strstream
```

Manipulator Descriptions

This section describes contents of the **iomanip.h** header file, which provides predefined manipulators, as well as support functions and classes that enable you to create your own manipulators.

class IOMANIP

Provide Manipulators

SYNOPSIS

```
#include <IOMANIP.h>

/* Macros for creating class names */
#define SMANIP(T)  ...
#define SAPP(T)    ...
#define IMANIP(T)  ...
#define IAPP(T)    ...
#define OMANIP(T)  ...
#define OAPP(T)    ...
#define IOMANIP(T) ...
#define IOAPP(T)   ...

// Start of IOMANIPdeclare macro

#define IOMANIPdeclare(T)

class SMANIP(T)
{
   public:
   SMANIP(T)(ios&(*f)(ios&, T), T d);
   friend istream& operator >>(istream& i,
         SMANIP(T)& m);
   friend ostream& operator <<(ostream& o,
         SMANIP(T)& m);
};

class SAPP(T)
{
public:
   SAPP(T)(ios&(*f)(ios&, T));
   SMANIP(T)operator()(T d);
};

class IMANIP(T)
{
public:
   IMANIP(T)(ios&(*f)(ios&, T), T d);
   friend istream& operator >>(istream& i,
         IMANIP(T)& m);
};

class IAPP(T)
{
public:
   IAPP(T)(ios&(*f)(ios&, T));
   IMANIP(T)operator()(T d);
};
class OMANIP(T)
{
public:
   OMANIP(T)(ios&(*f)(ios&, T), T d);
   friend ostream& operator <<(ostream& o,
         OMANIP(T)& m);
};
class OAPP(T)
{
public:
   OAPP(T)(ios&(*f)(ios&, T));
   OMANIP(T)operator()(T d);
};

class IOMANIP(T)
{
public:
   IOMANIP(T)(ios&(*f)(ios&, T), T d);
   friend istream& operator >>(istream& i,
         IOMANIP(T)&m);
   friend ostream& operator <<(ostream& o,
         IOMANIP(T)&m);
};

class IOAPP(T)
{
public:
   IOAPP(T)(ios&(*f)(ios&, T));
   IOMANIP(T)operator()(T d);
};
// End of IOMANIPdeclare macro

IOMANIPdeclare(int);
IOMANIPdeclare(long);

SMANIP(int) setw(int width);
SMANIP(int) setbase(int base);
SMANIP(int) setfill(int fill_char);
SMANIP(int) setprecision(int precision);
SMANIP(long) setiosflags(long flags);
SMANIP(long) resetiosflags(long flags);
```

DESCRIPTION

The **IOMANIP.h** header file declares some predefined manipulators, as well as support functions and classes that enable you to create your own manipulators. A *manipulator* is a value that can be used to effect some change to a stream by inserting it into or extracting it from the stream. For example, the **flush** function is a manipulator of **ostream** objects:

```
cout << flush; // Causes cout to be flushed.
```

In fact, any function of one the following types is a manipulator:

ostream& (ostream&)
is a manipulator for **ostream** objects.

istream& (istream&)
is a manipulator for **istream** objects.

ios& (ios&)
is a manipulator for **istream** or **ostream** objects.

You can also create manipulators that have arguments. The **IOMANIP.h** header file defines two manipulator-creation classes for each type of stream (**ios**, **istream** , **ostream** , and **iostream**). One class has a name in the form ***x*MANIP(T)** , and the other class has a name in the form ***x*APP(T)** , where **T** is an identifier that names a type (such as a **typedef** name for a class name) and *x* is a letter such as S.

For **ios** objects, these two classes are named **SMANIP(T)** and **SAPP(T)** .

PREDEFINED MANIPULATORS

The predefined manipulators defined by **iomanip.h** allow you to control various pieces of the format state of a stream. These manipulators are described in the following list.

SMANIP(int) setw(int w)
returns a manipulator (an **SMANIP(int)**) that can be used to set the **width()** value of an **ios** object.

SMANIP(int) setbase(int base)
returns a manipulator that can be used to set the default numeric conversion base of an **ios** object. The argument must be one of the values 8, 10, or 16.

SMANIP(int) setfill(int f)
returns a manipulator that can be used to set the **fill()** value of an **ios** object.

SMANIP(int) setprecision(int p)
returns a manipulator that can be used to set the **precision()** value of an **ios** object.

SMANIP(long) setiosflags(long flags)
returns a manipulator that can be used to set the **flags()** value of an **ios** object.

SMANIP(long) resetiosflags(long flags)
returns a manipulator that can be used to reset the **flags()** value of an **ios** object.

EXAMPLES USING PREDEFINED MANIPULATORS

The following example transmits **********27,00048** :

```
cout << setw(10) << setfill('*')
     << 27 << ',' << setw(5)
     << setfill('0') << 48;
```

The following example transmits **32,5** :

```
cout << setprecision(2) << 32.1 << ','
     << setprecision(0) << 5.3;
```

The following example sets the **skipws** bit in **cout**'s format state:

```
cout << setiosflags(ios::skipws);
```

The following example clears the **skipws** bit in **cout**'s format state:

```
cout << resetiosflags(ios::skipws);
```

USER-DEFINED MANIPULATORS

As well as the predefined manipulators described in the previous section, the **IOMANIP.h** header file also provides the means for you to create your own manipulators. It does this by defining a macro, **IOMANIPdeclare(T)** , that when invoked with a **typedef** name for **T** declares the following classes:

SMANIP(T) and SAPP(T)
use with **ios** objects.

IMANIP(T) and IAPP(T)
use with **istream** objects.

OMANIP(T) and OAPP(T)
use with **ostream** objects.

IOMANIP(T) and IOAPP(T)
use with **iostream** objects.

class SMANIP(T) and **SAPP(T)** are explained in detail in this section; the other classes are very similar to **SAPP(T)** and only the differences between them and **class SMANIP** and **SAPP(T)** are noted.

If you are going to create new manipulators using the various ***x*MANIP(T)** and ***x*APP(T)** classes, the classes must first be defined for a particular type name. This is done by putting the following definition in any module that uses the ***x*MANIP(T)** or ***x*APP(T)** classes for a particular type name:

```
IOMANIPdeclare(type-name);
```

where *type-name* can be any valid type identifier. Because **int** and **long** are the most commonly used type names in manipulators, the **IOMANIPdeclares** for these type names are included in the **IOMANIP.h** header file, and your program should not declare them again. If you need to create manipulators using the ***x*MANIP(T)** and ***x*APP(T)** classes for type names other than **int** or **long** , you must include a use of **IOMANIPdeclare()** in your module.

For example, before using ***x*MANIP(T)** and ***x*APP(T)** classes to create manipulators that accept **char** arguments, the ***x*MANIP(T)** and ***x*APP(T)** classes for the type name **char** must be declared as follows:

```
#include <IOMANIP.h>
IOMANIPdeclare(char);
```

If you need to create manipulators that accept arguments of more complicated types, like **char*** arguments, you must first declare a **typedef** for the type because **IOMANIPdeclare** requires a single-word type name. Here is an example:

```
#include <IOMANIP.h>
typedef char* STRING;
IOMANIPdeclare(STRING);
```

class SMANIP(T)

provides a constructor and two operators, as detailed next.

SMANIP(T)(ios&(*f)(ios&, T), T d)
constructs an **SMANIP(T)** and returns a single argument manipulator by collecting the function **f** and argument **d** into a single manipulator value. It is assumed that **f** is a function that changes **ios** in some way using the value of **d** .

friend istream& operator >> (istream& i, SMANIP(T)& m)

friend ostream& operator << (ostream& o, SMANIP(T)& m)
enable **SMANIP(T)** objects to be "inserted-into" **istream** objects and "extracted-from" **ostream** objects, respectively. They each use the values **f** and **d** from the **SMANIP(T)** value. They then call **f(myios,d)** where **myios** is the **ios** part of **i** or **o** , respectively. It is assumed that **f** is a function that changes **ios** in some way using the value of **d** .

It is often easier to create manipulators using the applicator classes, in this case **SAPP(T)** , than to use the ***x*MANIP(T)** classes. **class SAPP(T)** is described next.

class SAPP(T)

provides a constructor and an operator, as detailed next. **SAPP(T)** objects make it easier to use **SMANIP(T)** objects. The EXAMPLES section gives an example of using an **SAPP(T)** object. Here are the members of **class SAPP(T)** :

SAPP(T) (ios&(*f)(ios&, T))
initializes an **SAPP(T)** object to contain **f** .

SMANIP(T) operator() (T d)
creates and returns an **SMANIP(T)** object using the **f** from the **SAPP(T)** and the **d** argument.

Other manipulator classes

The rest of the classes defined by **IOMANIPdeclare(T)** are similar to class **SAPP(T)** , with the following differences:

□ for **IMANIP(T)** and **IAPP(T)** , **f** has type

```
istream&(*f)(istream&, T)
```

□ for **OMANIP(T)** and **OAPP(T)** , **f** has type

```
ostream&(*f)(ostream&, T)
```

□ for **IOMANIP(T)** and **IOAPP(T)** , **f** has type

```
iostream&(*f)(iostream&, T)
```

□ **IMANIP(T)** does not contain **operator <<**
□ **OMANIP(T)** does not contain **operator >>**

EXAMPLES OF USER-DEFINED MANIPULATORS

The following code creates a manipulator **setwidth** , which works like the library's **setw** .

```
ios& setw_func(ios& i, int w)
{
   i.width(w);
   return i;
}
SAPP(int) setwidth(setw_func);
```

CHAPTER

5

Debugging C++ Programs Using the SAS/C Debugger

Introduction

For the most part, debugging C++ programs is the same as debugging C programs. There are only a few differences, which are the focus of this chapter. This chapter does not attempt to teach you how to set up and use the basic features of the debugger. For general information about setting up the debugger under CMS and TSO and for tutorials on using the debugger, refer to the *SAS/C Debugger User's Guide and Reference, Third Edition*.

The main areas of concern in debugging C++ programs are

- specifying C++ function names in debugger commands
- specifying C++ expressions in debugger commands
- searching for data objects
- debugging initialization and termination routines
- setting breakpoints in dynamically loaded modules.

These areas are covered in the next few sections. The final section contains an example that illustrates debugging a sample C++ program.

Specifying C++ Function Names

One of the unique features of the SAS/C C++ Development System is that the debugger accepts and understands C++ function names, including multitoken and overloaded function names. This section describes how to specify C++ function names in debugger commands.

If you are specifying a nonoverloaded, single token function name in a debugger command, you do not have to do anything different from specifying a C function name. For example, you could issue the following command:

```
break func1 entry
```

There are additional rules, however, for specifying multitoken C++ function names and overloaded function names in debugger commands. This section also explains how to specify member function names and file-scope function names.

The rules for specifying constructor and destructor function names are unique to these types of functions and are covered separately. There is also a section explaining how the debugger handles function names when you mix C and C++ code and a section describing how the debugger handles translator-generated functions (such as assignment operators and copy constructors).

Multitoken Function Names

Multitoken function names are function names that are not just a C++ identifier, but that contain other items such as the scope operator (::) or two-word function names such as operator and conversion functions. Here are some examples of multitoken C++ function names:

- **`ABC::ABC`**
- **`myfunc::~myfunc`**
- **`operator int *`**
- **`ABC::operator >=`**

When you specify a multitoken C++ function name in a debugger command, the function name must be enclosed in double quotes. Here is an example of the **break** command and a multitoken function name. This command specifies to break at the entry to member function **func1** in **class ABC**:

```
break "ABC::func1" entry
```

Spaces around tokens that are not identifiers are optional.

Overloaded Function Names

One of the things that sets C++ apart from C is C++ support of overloaded functions. However, overloaded functions present a challenge for the debugger because the debugger has to determine which function you want to access.

When you specify an overloaded function name in a debugger command, you are presented with a numbered list of C++ function names with arguments. Determine which number represents the function you want to access, and reissue the debugger command by appending a parenthesized number after the function name. For example, suppose you have the following three constructors declared in **class myclass**, in this order:

```
myclass(char);
myclass(short);
myclass(long);
```

If you issue a **break "myclass::myclass" entry** command, the debugger shows you the following list:

```
1  myclass::myclass(char)
2  myclass::myclass(short)
3  myclass::myclass(long)
```

You can place a breakpoint on entry to the constructor that takes a **short** by specifying the following **break** command:

```
break "myclass::myclass"(2) entry
```

As long as you do not relink your program, the subscript numbers for overloaded functions remain the same. For example, you can define a debugger macro or alias using the subscripts and use it throughout your debugging session.

Instead of choosing a particular number, you can specify that the command apply to all instances of the function by using 0 as the parenthesized number. For example, the following command sets breakpoints on entry to any **myfunc** function in **class myclass**, regardless of the argument type:

```
break "myclass::myfunc"(0) entry
```

However, a subscript of 0 is valid only at entry hooks, return hooks, call hooks, or * (that is, all line hooks). The only commands that permit a subscript of 0 are **break, trace, ignore, runto**, and **on**.

File-Scope and Member Functions

The debugger uses the scope operator **(::)** to determine if you want to access either a filescope or a member function.

If you have declared both a filescope **myfunc** and a member function **myfunc** in **class ABC**, use the scope operator to tell the debugger which function you mean when you issue debugger commands:

"::myfunc"
refers to a file-scope function of name **myfunc**.

"ABC::myfunc"
refers to a member function of name **myfunc** in **class ABC**.

If you have only a file-scope function named **myfunc**, or only one member function named **myfunc** (but not both a file-scope function and a member function), you can omit the scope operator and specify just the function name in the debugger command.

Note: If the debugger is stopped in a member function when you issue a debugger command that includes only the function name (and no scope operator), the command works as if you were not stopped in a member function. That is, the debugger does not automatically prefix the function name with the class name of the class whose member function you are stopped in. This is slightly different from the behavior for data objects, where the class name is automatically prefixed (see “Searching for Data Objects” on page 100). △

Constructors and Destructors

When you specify a constructor or destructor in a debugger command, it must be in one of the following two forms:

" ***class-name::class-name***"
indicates a constructor.

" ***class-name::~class-name***"
indicates a destructor.

Here is an example of setting a breakpoint on entry to the destructor for **class ABC**:

```
break "ABC::~ABC" entry
```

Functions in a Mix of C and C++ Code

If your load module contains at least one C++ compilation, your load module also contains a list of all function names visible to C++ compilations. If you issue a debugger command that refers to a function name not in this list, the debugger issues a warning message and assumes the function is a C function.

For example, suppose you have the following construct, where **a()** is a C++ function and **b()** and **c()** are C functions:

a() calls **b()** calls **c()**

There is a function prototype for **b()** in the compilation containing **a()**. Because **b()** is visible to a C++ compilation, it is contained in the debugger’s list of visible function names. But no function prototype for **c()** is visible in any C++ compilation. Therefore, **c()** is not contained in the list of function names visible to the debugger. If you use **c()** in a debugger command (such as in a **break** command), the debugger issues a message that it cannot resolve the function name using the debugger file. The debugger therefore assumes that **`c()`** is a C function.

Note: If you see the warning message about unresolved function names yet you know that your program consists of only C++ functions, check the spelling of function names in your debugger commands. △

Translator Generated Functions

The translator creates a number of functions automatically. These functions are required by the C++ language and follow the usual C++ rules. The functions that the translator may create include constructors, copy constructors, assignment operators, and destructors. The translator creates such a function when there is not a user-defined version of the function. The list of overloaded constructors that is displayed by the debugger when you issue a debugger command may include a translator generated constructor as well as the user-defined constructors.

The following list shows the declarations for translator generated functions:

`class::class()`
: is the default constructor for class ***`class`***. This constructor is called whenever an object of type class is defined without an explicit initializer.

`class::class(const|volatile class&)`
: is the default copy constructor for class ***`class`***. A default copy constructor is created for any **`class`**, **`struct`**, or **`union`** that does not have a user-defined copy constructor. The copy constructor is called to initialize an object of type **`class`** with another object of the same class. The presence of **`const`** or **`volatile`** depends on the characteristics of the class.

`class& class::operator=(const|volatile class&)`
: is the default assignment operator. This operator is called when an object of type **`class`** has an object assigned to it. The presence of **`const`** or **`volatile`** depends on the characteristics of the class.

`class::~class()`
: is the default destructor. This destructor is called when an object of type **`class`** goes out of scope.

You may occasionally step into one of these translator-generated functions as you debug your code. When this happens, the Source window displays the source text at the class definition and the Status window displays the function name.

Specifying Expressions

The debugger supports the use of operators, types, and casts that are specific to C++. This section delineates these items and explains how expressions are evaluated for C++ programs in the debugger. Note that only standard C++ operators are supported in expressions. That is, user-defined overloaded operators cannot be used. This includes the use of complex and I/O stream operators. For example, you cannot specify **`print (a+b)`** where **`a`** and **`b`** are complex.

Operators

The following operators are supported in debugger expressions:

`::` (unary scope)
: indicates the scope operator (identifies the object or function as file-scope).

`::` (binary scope)
: indicates the scope operator (identifies the object or function as a member of a class).

`->*`
: indicates a member-pointer.

`.*`
: indicates a member-pointer.

Only one level of **`::`** is supported after a **`.`** or **`->`** operator. For example, the following is not valid syntax in a debugger command:

```
p->A::BB::c
```

Casts

In addition to the syntax for casts supported for C in the debugger, the keyword **`class`** is supported as in

```
(class TAG*)ADD
```

Casts to reference types are not supported. If two classes you are referencing are related (that is, one is derived from the other), the debugger performs the cast and issues a message that indicates address translation may have occurred.

Data Types

All debugger commands that support expressions support the following C++ data types:

- pointers to base or derived classes
- member-pointers
- references
- classes

The following sections detail any special considerations for using debugger commands with C++ data types. Static members of class objects do not participate in any **`assign`**, **`copy`**, **`dump`**, **`monitor`**, **`print`**, **`return`**, or **`watch`** commands that deal with objects of type **`class`**. For example, because all classes share the same **`static`** data, if you copy a class with the **`copy`** command, you do not modify static members.

A member-pointer is not considered a pointer in the C or C++ sense. Therefore, it is invalid to specify an expression of type member-pointer in a command (such as **`dump`**) that takes an address for an operand.

assign command

The **assign** command can be used to assign a pointer to an object of a derived class to a pointer to the base class. When multiple inheritance is used, this can cause the values of the derived pointer and the base pointer as printed by the **print** command to differ. This can also occur for assignments involving member-pointers.

An assignment to a reference assigns to the referenced object.

An assignment to a class object is permitted using an initializer list or a class object only if the class does not have base classes and if no user-defined constructors need be invoked to initialize the class object.

dump command and Dump window

The **dump** command and the Dump window support all the data types listed earlier in this section. The **dump** command takes an address as an argument and, by default, dumps memory corresponding to the size of the object. A member pointer can be one of two sizes, depending on whether it is a data pointer or a function pointer:

data pointers	are 4 bytes long.
function pointers	are 12 bytes long.

A member-pointer is not considered to yield an address type. Therefore the following command is not valid:

```
dump member-pointer
```

The following command dumps the referenced object, as explained in “Expression Evaluation” on page 100:

```
dump reference-object
```

monitor command

You cannot monitor an object through a reference variable. You can only monitor the reference variable itself. If you set a monitor on a class object that con- tains a reference, the storage allocated for the class object (which includes the storage allocated to the reference variable) is monitored. That is, the referenced object is not monitored.

For objects of derived classes, the base objects are also monitored. Because the entire storage of an object is monitored, it is possible for the monitor to be triggered without any change in the printed value. That is, for some data types (such as function member-pointers), some parts of the value may not be reflected in the value produced by the **print** command. The debugger attempts to detect corruption of control data and hidden pointers to virtual base classes (that is, members of a class that are in addition to those members you have explicitly created). If your program is erroneously overwriting memory, it could overwrite some of these control data or hidden pointers. Even though this corruption does not affect the value that is printed, the monitor is triggered.

return command

For a function that returns a reference, the **return** command returns a reference. A function may return a class using an initializer list or a class object only if the class does not have base classes and if no constructors are needed to initialize the class.

transfer command

For a reference type object, use the C++ notation for references (such as **class myobject&**) in the **typeof** keyword of the **transfer** command.

whatis command

The **whatis** command uses the C++ notation for references (such as **myobject&**) in its output.

For classes, **whatis** displays the following information, in addition to the usual C information:

- base classes
- access attributes
- all non-C member types that can occur as class members (classes, **enums**, **typedefs**, and functions, including function prototypes).

Expression Evaluation

Normal C++ rules are followed in expression evaluation, with the following exception. If a **static** member is dereferenced using the **->** or **.** operator, the expression to the left of the **->** or **.** operator is evaluated by the debugger.

Searching for Data Objects

Normal C++ scoping rules apply to most searches in the debugger. When you are in a member function, you do not have to specify **this->** to access class members. If your source code is structured so that classes are nested within classes, the debugger also searches any lexically enclosing scopes. The debugger applies normal C++ rules for ambiguity resolution.

You can access **static** members using the ***class-name*::*member-name*** syntax.

Debugging Initialization and Termination

Functions

A typical C++ program contains intialization functions in each compilation at program startup, which are called at program startup to initialize **static** and **extern** data defined in that compilation. If you want to debug one of these initialization functions, you can set a breakpoint on **__init_*sname***, where ***sname*** is the sname for that compilation. (See “Option Descriptions” on page 38.) Similarly, a function called **__term_*sname*** is called for each compilation at termination.

By default, the first function name shown in the Status window is one of these initialization functions. While the debugger is stopped in these functions, you can debug the initialization of **static** and **extern** variables. As you step through the initialization functions, each function is in turn shown in the Status window. Note that the initialization and termination functions are not shown in the Source window, as they do not exist in user C++ code.

Note: The functions **`__init_`*`sname`*** and **`__term_`*`sname`*** are implementation-dependent. Either the implementation or the names, or both, may change in a future release. △

Bypassing Initialization Functions

If you do not want to debug your program's initialization functions, you can bypass them in one of several ways:

- □ set breakpoints in functions that are of interest and issue a **`go`** command
- □ put a **`Break main entry`** command in your debugger profile and issue a **`go`** command (this causes debugging to begin at the **`main`** function)
- □ issue a **`Runto main entry`** command from the command line (this causes your program to advance to **`main`**).

Setting Breakpoints in Dynamically Loaded C++ Modules

The debugger keeps track of load modules containing C++ code as they are loaded and unloaded. When you specify a function in a debugger command, the debugger first looks for the function in the current load module. If it fails to find it there, the debugger searches the list of modules that have been loaded. If you want to set a breakpoint in a module that has not yet been loaded, you can set a breakpoint on entry to **`_dynamn`**. Then, when you reach this breakpoint, set the desired breakpoint in the module. Note, however, that **`_dynamn`** is called only after constructors for static and extern objects in the loaded module have been run.

C++ Debugging Example

This section provides an example of running the debugger with a C++ program. Some of the features the example illustrates include

- □ the need to enclose most C++ function names in double quotes.
- □ how to specify overloaded C++ function names using a subscript.
- □ how to specify all overloaded functions at once using a subscript of 0.
- □ how to set breakpoints in special C++ functions, such as constructors and destructors. (Specifically, this example shows that user-defined constructors and destructors can be debugged even when they are driven by the C++ **`new`** and **`delete`** operators.)
- □ how to debug C++ class member functions.

The important thing to remember is that debugging C++ programs with the SAS/C Debugger is virtually the same as debugging C programs. The debugger looks and feels the same, and in general the commands are the same. This example simply shows the few extra things to remember when you are debugging C++ programs. Once you complete this example, you should be ready to debug your own C++ programs.

Note: This example requires you to allocate the DDname DBGSLIB to the location of your standard header files. See *SAS/C Compiler and Library User's Guide* for more information on the DBGSLIB DDname. △

Example Source Code

Here is the source code for the example. The program declares **`class X`**, which includes one data object, two constructors, one member function, and a destructor. The member function multiplies the data object by 2. The destructor checks the value of the data object and prints an error message if the data object is not between 7 and 10. The **`main`** function uses the constructors to create several instances of class X, calls the member function four times (once for each instance of **`class X`**), and then deletes the instances of **`class X`**.

```
#include <iostream.h>
class X
{
public: int i;
      // first X constructor
   X(int ia)
   {
      i = ia;
   };
      // second X constructor
   X(int ib, int jb)
   {
      i = ib + jb;
   };
       // X member function myfunc()
   void myfunc()
   {
      i *= 2;
   };
      // X destructor
   ~X()
   {
      int id;
      id = i / 2;
      if (id < 7 || id > 10)
      printf("Error - Out of range\n");
   };
};
int main()
{
   X *x1 = new X(7);
   X *x2 = new X(6,2);
   X *x3 = new X(9);
   X *x4 = new X(9,1);
   x1->myfunc();
   x2->myfunc();
   x3->myfunc();
   x4->myfunc();
```

```
      delete x1;
      delete x2;
      delete x3;
      delete x4;
      return 0;
   }
```

In this example debugger session, you use the **break, go, query, drop**, and **on** debugger commands to complete the program. The numbered steps tell you what to type in and show the results of your commands in the various debugger windows.

You first need to translate, link, and run the sample program. Be sure to specify the **debug** option when you translate. For information on translating, linking, and running a program in your environment (TSO or CMS), see Chapter 2, "Using the SAS/C C++ Development System under TSO, CMS, MVS Batch, and OpenEdition," on page 13. For information on setting up your files and invoking the debugger, see the *SAS/C Debugger User's Guide and Reference, Third Edition* and the *SAS/C Compiler and Library User's Guide*. Display 5.1 on page 102 shows the appearance of the debugger at the beginning of your debugging session.

Enter the following debugger commands in sequence. Debugger commands are entered after the **Cdebug:** prompt located in the Command window at the bottom of the screen. (In the following discussion, debugger commands are shown in **monospace** font.)

1 **break "X::X"(0) entry**

This command sets breakpoints at entry to all constructors for **X**. The subscript of 0 is necessary because the constructors are overloaded functions. Because the function name has a multitoken name, double quotes are necessary.

2 **break main 37**
go
These two commands first set a breakpoint at line 37 of the **main** function and tell the debugger to advance to the first breakpoint. The debugger stops on line 37 of **main**, as shown in the illustration of the Source window in Display 5.2 on page 103.

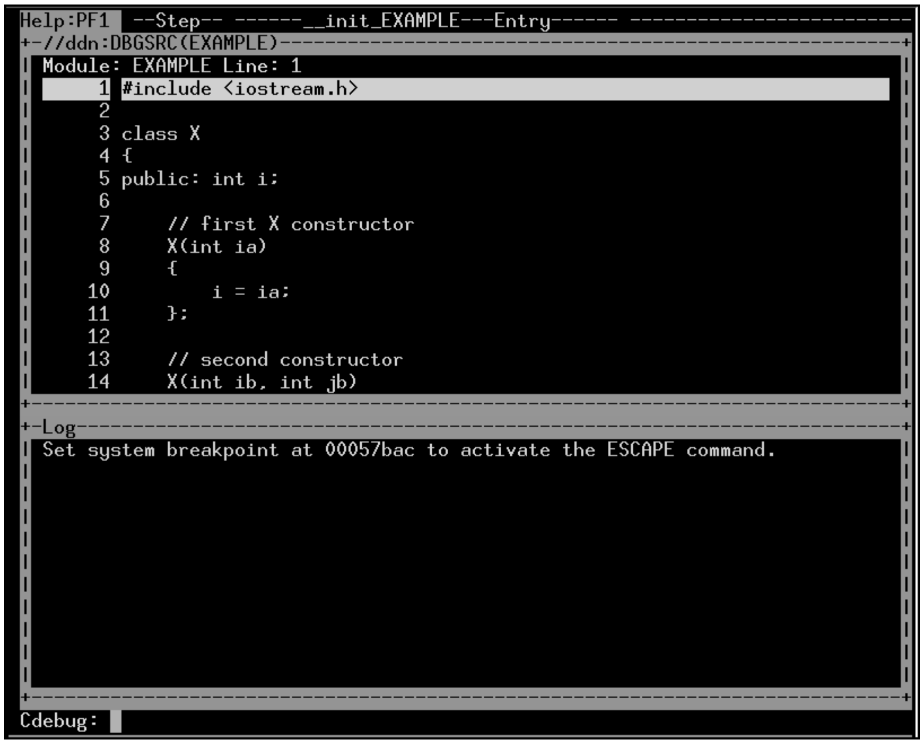
3 **go**
The program proceeds until it enters the first constructor. Display 5.3 on page 103 shows the Source window.

4 **go**
Yet another **go** causes the debugger to stop at entry to the next constructor, as shown in Display 5.4 on page 104.

5 **query**
This command requests the Log window to show all actions and monitors in effect. Two breakpoints are shown; of special interest is the first breakpoint, which shows the subscript of 0, indicating a breakpoint is in effect for all instances of the overloaded constructor, X. Display 5.5 on page 104 shows the Log window.

6 **drop 1**
break "X::X"(1) return

Display 5.1 Initial Appearance of the SAS/C Debugger

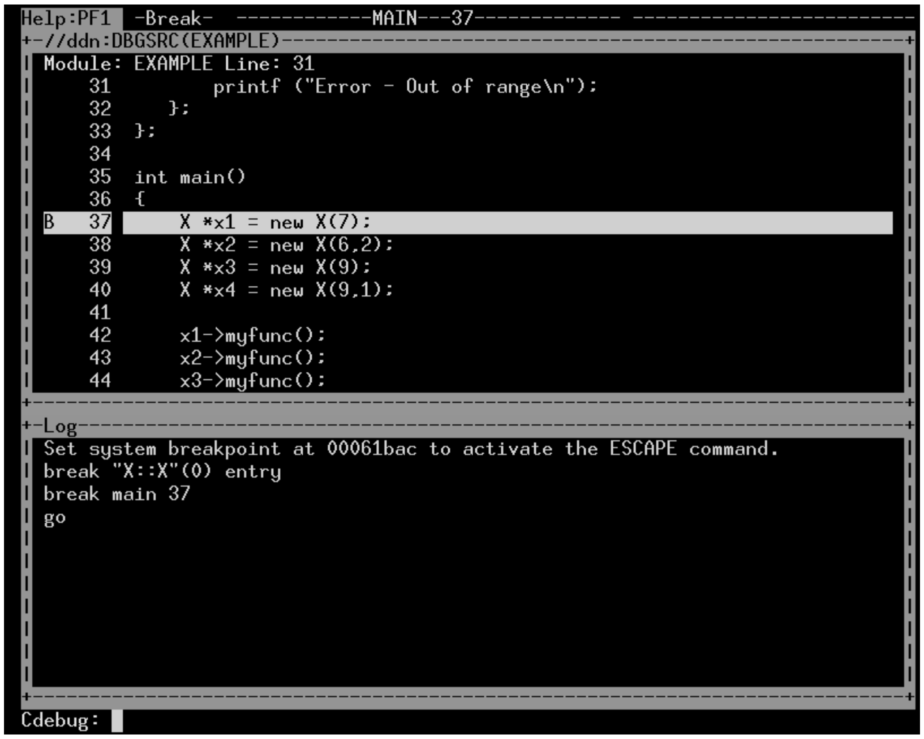


```
Help:PF1  --Step-- ------__init_EXAMPLE---Entry------ ------------------------
+-//ddn:DBGSRC(EXAMPLE)-------------------------------------------------------+
| Module: EXAMPLE Line: 1                                                      |
|       1 #include <iostream.h>                                                |
|       2                                                                      |
|       3 class X                                                              |
|       4 {                                                                    |
|       5 public: int i;                                                       |
|       6                                                                      |
|       7     // first X constructor                                           |
|       8     X(int ia)                                                        |
|       9     {                                                                |
|      10         i = ia;                                                      |
|      11     };                                                               |
|      12                                                                      |
|      13     // second constructor                                            |
|      14     X(int ib, int jb)                                                |
+------------------------------------------------------------------------------+
+-Log--------------------------------------------------------------------------+
| Set system breakpoint at 00057bac to activate the ESCAPE command.            |
|                                                                              |
+------------------------------------------------------------------------------+
Cdebug:
```

```
break "X::X"(2) return
on myfunc entry print i
 on myfunc return print i
 b "X::~X" return
query
```

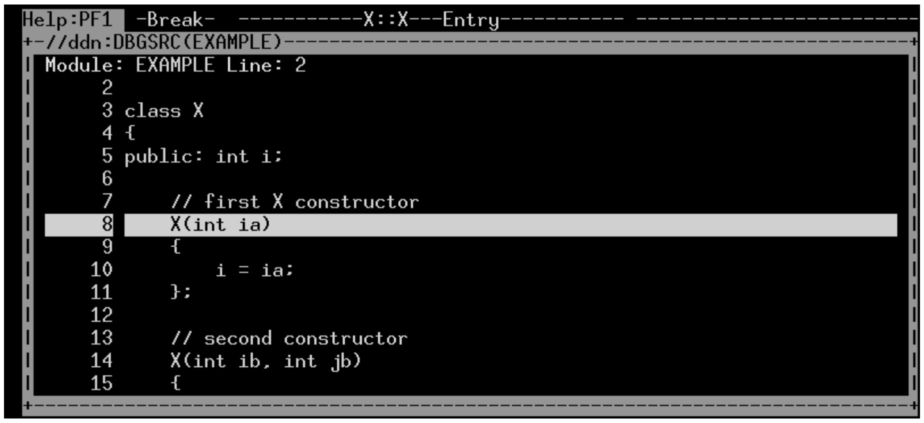
The **drop** command drops breakpoint #1 (it is no longer necessary). Next, you set breakpoints on the return of both versions of the overloaded constructor, using the subscripts 1 and 2. The two **on** commands tell the debugger to print the value of the **X** member function **i** on the entry to and return from the **myfunc** member function. The next command sets a breakpoint on the return of the destructor (**b** is an abbreviation for **break**). Finally, the **query** command shows all the actions and monitors in effect. Of special interest are breakpoints #3 and #4. These breakpoints show the prototypes for the first and second constructors (the first takes a single **int**; the second takes two **int**s). Display 5.6 on page 104 shows the Log window after this series of commands.

Display 5.2 Stopping at Main



```
Help:PF1  -Break-  ------------MAIN---37------------ ------------------------
+-//ddn:DBGSRC(EXAMPLE)----------------------------------------------------+
| Module: EXAMPLE Line: 31                                                  |
|     31         printf ("Error - Out of range\n");                         |
|     32     };                                                             |
|     33 };                                                                 |
|     34                                                                    |
|     35 int main()                                                         |
|     36 {                                                                  |
| B   37     X *x1 = new X(7);                                              |
|     38     X *x2 = new X(6,2);                                            |
|     39     X *x3 = new X(9);                                              |
|     40     X *x4 = new X(9,1);                                            |
|     41                                                                    |
|     42     x1->myfunc();                                                  |
|     43     x2->myfunc();                                                  |
|     44     x3->myfunc();                                                  |
+--------------------------------------------------------------------------+
+-Log----------------------------------------------------------------------+
| Set system breakpoint at 00061bac to activate the ESCAPE command.        |
| break "X::X"(0) entry                                                    |
| break main 37                                                            |
| go                                                                       |
+--------------------------------------------------------------------------+
Cdebug:
```

Display 5.3 Stopping in a Constructor



```
Help:PF1  -Break-  -----------X::X---Entry----------- ------------------------
+-//ddn:DBGSRC(EXAMPLE)----------------------------------------------------+
| Module: EXAMPLE Line: 2                                                   |
|      2                                                                    |
|      3 class X                                                            |
|      4 {                                                                  |
|      5 public: int i;                                                     |
|      6                                                                    |
|      7     // first X constructor                                         |
|      8     X(int ia)                                                      |
|      9     {                                                              |
|     10         i = ia;                                                    |
|     11     };                                                             |
|     12                                                                    |
|     13     // second constructor                                          |
|     14     X(int ib, int jb)                                              |
|     15     {                                                              |
+--------------------------------------------------------------------------+
```

7 go

This **go** command causes the debugger to proceed until it reaches the return from the second constructor.

8 go

Another **go** causes the debugger to proceed until it reaches the return from the first constructor.

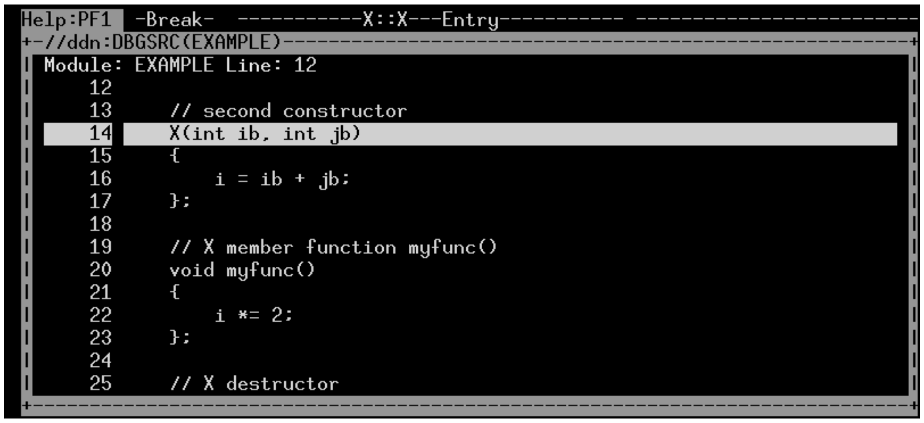
9 go

This **go** command causes the debugger to stop again at return from the second constructor.

go

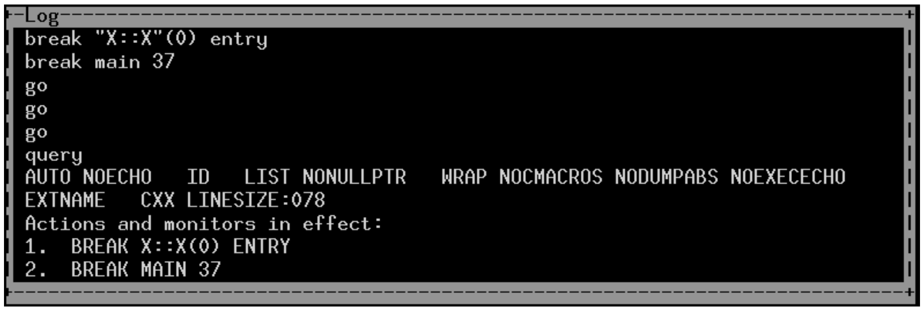
After this **go** command, the Log window shows the output from eight print commands (4 from entry to **myfunc** and 4 from return from **myfunc**). The values printed are 7, 14, 8, 16, 9, 18, 10, and 20. Notice also that the debugger stops at return from the destructor.

Display 5.4 Stopping in Another Constructor



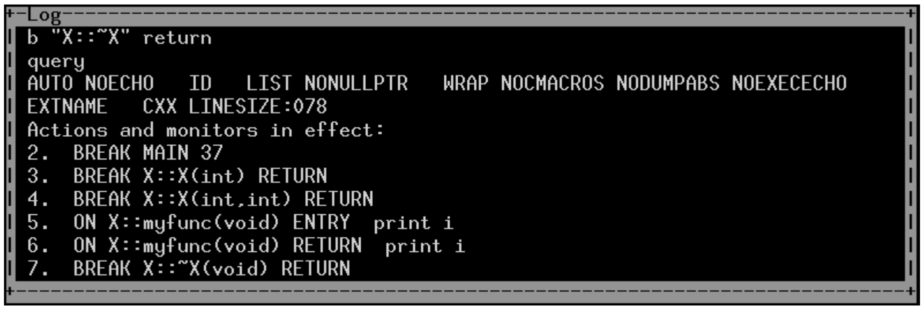
```
Help:PF1  -Break-  -----------X::X---Entry----------- -----------------------
+-//ddn:DBGSRC(EXAMPLE)----------------------------------------------------+
| Module: EXAMPLE Line: 12
     
     // second constructor
     X(int ib, int jb)
     {
         i = ib + jb;
     };
     
     // X member function myfunc()
     void myfunc()
     {
         i *= 2;
     };
     
     // X destructor
```

Display 5.5 Querying the Log Window



```
+-Log----------------------------------------------------------------------+
 break "X::X"(0) entry
 break main 37
 go
 go
 go
 query
 AUTO NOECHO   ID   LIST NONULLPTR   WRAP NOCMACROS NODUMPABS NOEXECECHO
 EXTNAME   CXX LINESIZE:078
 Actions and monitors in effect:
 1.  BREAK X::X(0) ENTRY
 2.  BREAK MAIN 37
```

Display 5.6 Another Log Window



```
+-Log----------------------------------------------------------------------+
 b "X::~X" return
 query
 AUTO NOECHO   ID   LIST NONULLPTR   WRAP NOCMACROS NODUMPABS NOEXECECHO
 EXTNAME   CXX LINESIZE:078
 Actions and monitors in effect:
 2.  BREAK MAIN 37
 3.  BREAK X::X(int) RETURN
 4.  BREAK X::X(int,int) RETURN
 5.  ON X::myfunc(void) ENTRY  print i
 6.  ON X::myfunc(void) RETURN  print i
 7.  BREAK X::~X(void) RETURN
```

Display 5.7 on page 105 shows the Log and Source windows for this step.

10 go

go

go

At each of these **go** commands, the debugger stops at the return from the destructor.

11 `break "X::Y" entry`

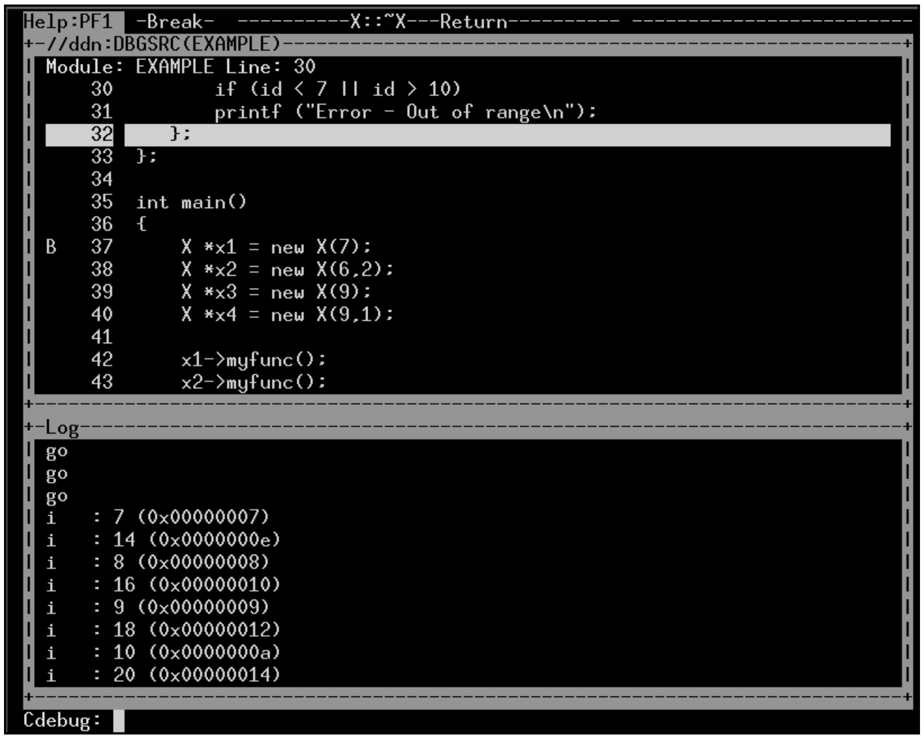
This command illustrates the warning message issued by the debugger when it cannot find a function in the debugger's list of visible functions. Because **`X::Y`** is not a valid function name for this program, a warning message appears in the Log window, as shown in Display 5.8 on page 105.

See "Functions in a Mix of C and C++ Code" on page 98 for more information on when you may see this warning message.

12 go

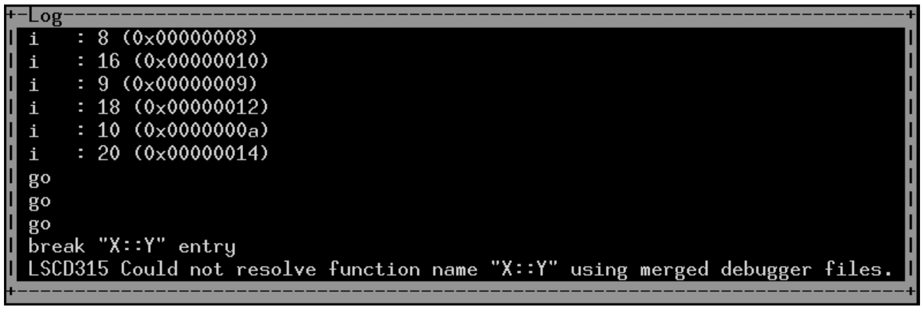
With this last **go** command, execution completes, and the debugger terminates.

Display 5.7 Log Window Showing Results from Print Commands



```
Help:PF1  -Break-  ----------X::~X---Return---------- ------------------------
+-//ddn:DBGSRC(EXAMPLE)----------------------------------------------------+
| Module: EXAMPLE Line: 30                                                 |
|      30        if (id < 7 || id > 10)                                    |
|      31        printf ("Error - Out of range\n");                        |
|      32     };                                                           |
|      33  };                                                              |
|      34                                                                  |
|      35  int main()                                                      |
|      36  {                                                               |
| B    37      X *x1 = new X(7);                                           |
|      38      X *x2 = new X(6,2);                                         |
|      39      X *x3 = new X(9);                                           |
|      40      X *x4 = new X(9,1);                                         |
|      41                                                                  |
|      42      x1->myfunc();                                               |
|      43      x2->myfunc();                                               |
+--------------------------------------------------------------------------+
+-Log----------------------------------------------------------------------+
| go                                                                       |
| go                                                                       |
| go                                                                       |
| i   : 7 (0x00000007)                                                     |
| i   : 14 (0x0000000e)                                                    |
| i   : 8 (0x00000008)                                                     |
| i   : 16 (0x00000010)                                                    |
| i   : 9 (0x00000009)                                                     |
| i   : 18 (0x00000012)                                                    |
| i   : 10 (0x0000000a)                                                    |
| i   : 20 (0x00000014)                                                    |
+--------------------------------------------------------------------------+
Cdebug:
```

Display 5.8 Warning Message Displayed when the Debugger Can't Find a Function Name



```
+-Log----------------------------------------------------------------------+
| i   : 8 (0x00000008)                                                     |
| i   : 16 (0x00000010)                                                    |
| i   : 9 (0x00000009)                                                     |
| i   : 18 (0x00000012)                                                    |
| i   : 10 (0x0000000a)                                                    |
| i   : 20 (0x00000014)                                                    |
| go                                                                       |
| go                                                                       |
| go                                                                       |
| break "X::Y" entry                                                       |
| LSCD315 Could not resolve function name "X::Y" using merged debugger files. |
+--------------------------------------------------------------------------+
```

PART 2

Appendices

APPENDIX

1

Converting a C Program to a C++ Program

Introduction

C++ was intended to be as close to C as possible. However, C++ added many features to the language. Therefore, valid C programs are not always valid C++ programs. Examples of the differences between C and C++ are extra reserved keywords and required function prototypes. If you want to convert your C programs to C++ programs, you should be aware of the considerations outlined in this appendix.

Differences between C and C++

The following sections outline the major differences between C and C++ and describe how to change the C constructs to their corresponding C++ constructs.

Reserved Keywords

The following keywords cannot be used as identifiers in C++ programs because they are reserved:

asm	**new**	**this**
catch	**operator**	**throw**
class	**private**	**try**
const_cast	**protected**	**typeid**
delete	**public**	**typename**
dynamic_cast	**reinterpret_cast**	**virtual**
friend	**static_cast**	
inline	**template**	

If these keywords are used as identifiers in your C program, you must change them to some other name.

Function Prototypes

In C++, a function must have a prototype in scope when it is called. In C, this is not necessary, although it is regarded as good programming practice. It is common in C code to find function declarations that are not prototypes (because frequently it is only the return type that is important to declare). For example, the following statement is regarded in C as a declaration but not a prototype of a function with no parameters:

```
int myfunc();
```

In C++, this same statement is interpreted as a prototype equivalent to the following:

```
int myfunc(void);
```

If you mistakenly use the **myfunc()** form in your C++ program as a prototype and then call the function with one or more parameters, you will receive error messages. In converting a program from C to C++, you must change all of your nonprototypical function declarations to prototypes.

"C" Linkage

If a C++ function calls C functions that are not to be converted to C++ functions (using the guidelines in this appendix), the declarations of these C functions in the C++ source (or header files) must include the extern "C" qualifier or be within an extern "C" block. For the standard SAS/C library functions, "C" linkage is specified in the SAS/C header files.

Multiple Declarations of File-Scope Variables

In ANSI C, a file-scope variable can be declared more than once without using the **extern** keyword. This is not legal in C++. To convert your program from C to C++, make sure that all but one declaration of a file-scope variable has the **extern** keyword. If the file-scope variable is initialized, the initializer should be attached to the one declaration without the **extern** keyword.

Assigning Integers to enums

In C++, an integer cannot be assigned directly to an **enum** as it could be in C. If your program contains such assignments, first cast the integer to the **enum** type, as in the following example:

```
int i;
enum X {a, b, c} e;

e = i; /* Legal in C, not in C++. */

e = (enum X)i; // Legal in C++.
```

Of course, the best approach is to use the correct **enum** constant in the first place.

Assigning void* Values to Pointers

In C++, a **void*** pointer can be assigned only to another **void*** pointer. If your C program contains assignments of **void*** pointers to other kinds of pointers, you should cast the **void*** to the other pointer type before the assignment, as in the following example:

```
int* ip;
void* vp;
ip = vp; /* Legal in C, not in C++. */
ip = (int*)vp; // Legal in C++.
```

Using Strings to Intitialize a Character Array

In C++, a string used to initialize a character array must be at least one character shorter than the array it is initializing. The extra element is for the terminating null character. The following example shows how this works:

```
char a[3] = "SAS"; /* Legal in C, not in C++. */

   // Legal in C++, keeps the terminating null
   // character.
char a[4] = "SAS";

   // Legal in C++, omits the null-character.
char a[3] = {'S', 'A', 'S'};
```

Using File-Scope Constants

In C, file-scope constants are external by default. In C++, file-scope constants are **static** by default. You can take two approaches to converting this type of problem:

- If the constant is a large array or structure, add **extern** to the declaration of the file-scope constant that has the initialization.
- If the constant is not large, move the declaration of the constant with the initialization into a header file included by all modules that need access to the constant. Then remove any **extern** keywords attached to declarations of the constant.

Using #pragma linkage

The SAS/C extension **#pragma linkage** is supported only within **extern "C"** linkage blocks in C++ programs. Programs that use the **#pragma linkage** feature outside **extern "C"** blocks should be changed to use the **__ibmos** keyword.

To change uses of **#pragma linkage**, remove the **#pragma linkage** and add **__ibmos** to the declaration of the function that needs IBM OS linkage. The following example shows equivalent C and C++ usage:

```
   /* Meaningful only in C. */
#pragma linkage (fname, OS)
int fname();

   // Meaningful in both C and C++.
__ibmos int fname();
```

__ibmos functions always have "C" linkage. For more information on the **__ibmos** keyword, refer to the *SAS/C Compiler and Library User's Guide, Fourth Edition.*

Identical Class Names and typedef Names

In C++, if a class name and a **typedef** name are the same, the **typedef** name must refer to the class. If your C program contains a class (structure) name and a **typedef** name that are the same, but refer to different things, you must change one of the names. Here is an example:

```
  /* This code is legal in C, but not in C++,
     because VAL refers to an int and a struct. */
typedef int VAL;
struct VAL
{
long x;
char *text;
};

  // This code is legal in both C and C++,
  // because VAL refers to the same thing in each
  // case.
struct VAL
{
```

```
long x;
char *text;
};

typedef struct VAL VAL;
```

Embedded Structure Tags

In C, a structure tag can be defined inside the definition of some other structure tag. This is also legal in C++, but because structures (classes) are scopes in C++, the inner structure tag is hidden inside the outer structure. If you do not want this scoping effect, do not embed structure tags. (In other words, move the inner structure's definition outside the outer structure's definition, or use the scope operator (**`::`**) to use the inner structure's tag in subsequent instantiations, as in **`outer::inner`**.)

Using goto Statements

C++ does not allow you to use the **`goto`** statement to jump over an initialization. If your C program contains such jumps, you must remove them.

Differing Types

C++ defines the types of character literals and enumerations differently than C does.

Character literals

In C, the type of a single-character character literal (for example, ' **`H`**') is **`int`**. In C++, character literals are of type **`char`**. You do not usually have to worry about this distinction, unless your program expects the result of **`sizeof('H')`** to be greater than one.

Enumerations

In C, the type of an **`enum`** is **`int`**. In C++, enumerations have type **`enum`**. Again, you do not usually have to worry about this distinction; just remember that the size of an **`enum`** may be different in C and in C++.

APPENDIX 2

Header Files, Classes, and Functions

Introduction

Chapter 4, "Standard Libraries," on page 47 describes in detail the classes and functions supplied with the SAS/C C++ Development System. This appendix provides a cross-reference to the information contained in that chapter.

Language Support

Table A2.1 on page 113 provides a reference for the classes and functions in the **`new.h`** and **`typeinfo.h`** header files. This information includes header filenames, the classes contained in those header files, and the functions contained in those classes.

Table A2.1 Header Files, Classes, and Functions for Language Support

Header File	Class	Functions

`new.h`		**`operator delete()`**
		`operator new()`
		`set_new_handler()`

`typeinfo.h`	**`type_info`**	**`operator ==()`**
		`operator !=()`
		`before()`
		`name()`
		`set_terminate()`
		`terminate()`

Complex Library Contents

Besides two constructors, the following functions are provided by **`class complex`**, which constitutes the complex library. All these functions are public and are contained in the header file **`complex.h`**. The functions are listed in the order in which they appear in Chapter 4.

```
abs()                    arg( )
conj()                   imag()
norm()                   polar()
real()

exp()                    log( )
pow()                    sqrt( )

sin()                    cos ( )
sinh()                   cosh ( )

operator +()             operator- ( )
operator *()             operator / ( )
operator ==)             operator != ( )
operator +=( )           operator -= ( )
operator *=              operator /=( )

operator << ( )          operator >> ( )
```

Streams Library Contents

The tables in this section give information about the stream and buffer classes supplied with the streams library. This information includes header filenames, the classes contained in those header files, and the functions contained in those classes. It also lists the base classes for derived classes. The header files, classes within header files, and member functions are listed alphabetically. The constructors and destructors are not listed in the function columns because they are implied.

Stream Classes

Table A2.2 on page 114 gives the header files, classes, and functions for the stream classes.

Table A2.2 Header Files, Classes, and Functions for Streams

Header File	Class (Base Classes)	Functions
bsamstr.h	**bsamstream (iostream)**	**add_member() clear_error_info() close() dcbblksize() dcblrecl() dcbrecfm() delete_member() error_info() find() getdcb() get_ddname() get_member() get_user_data() init_directory() open() rdbuf() rename_member() replace_member() setbuf() set_user_data() stow()**
	ibsamstream (istream)	**clear_error_info() close() dcbblksize() dcblrecl() dcbrecfm() error_info() find() getdcb() get_ddname() get_member() get_user_data() open() rdbuf() setbuf() set_user_data()**
	obsamstream (ostream)	**add_member() clear_error_info() close() dcbblksize() dcblrecl() dcbrecfm() delete_member() error_info() find() getdcb() get_ddname() get_member() get_user_data() init_directory() open() rdbuf() rename_member() replace_member() setbuf() set_user_data() stow()**
fstream.h	**fstream (iostream)**	**close() open() rdbuf() setbuf()**
	ifstream (istream)	**close() open() rdbuf() setbuf()**
	ofstream (ostream)	**close() open() rdbuf() setbuf()**
iomanip.h	**IOMANIP**	See the class description
iostream.h	**ios**	**bad() bitalloc() clear() eof() fail() fill() flags() good() iword() operator void*() operator!() precision() pword() rdbuf() rdstate() setf() tie() unsetf() width() xalloc()**
	iostream (ostream) (istream)	defines nothing but constructors

Header File	Class (Base Classes)	Functions
	istream **(ios)**	**gcount()** **get()** **getline()** **ignore()** **ipfx()** **operator>>()** **peek()** **putback()** **read()** **seekg()** **sync()** **tellg()** **ws()**
	ostream **(ios)**	**endl()** **ends()** **flush()** **operator<<()** **opfx()** **osfx()** **put()** **seekp()** **tellp()** **write()**
	streampos	**fpos()** **operator long()**
stdiostream.h	**stdiostream** **(iostream)**	**rdbuf()** **stdiofile()**
strstream.h	**istrstream** **(strstreambuf)** **(istream)**	**rdbuf()**
	ostrstream **(strstream-base)** **(ostream)**	**pcount()** **rdbuf()** **str()**
	strstream **(iostream)**	**pcount()** **rdbuf()** **str()**

Buffer Classes

Table A2.3 on page 115 gives the header files, classes, and functions for the buffer classes.

Table A2.3 Header Files, Classes, and Functions for Buffers

Header File	Class (Base Classes)	Functions
bsamstr.h	**bsambuf** **(streambuf)**	**add_member()** **attach()** **clear_error_info()** **close()** **dcbblksize()** **dcblrecl()** **dcbrecfm()** **delete_member()** **error_info()** **find()** **getdbc()** **get_ddname()** **get_member()** **get_user_data()** **init_directory()** **is_open()** **open()** **rename_member()** **replace_member()** **setbuf()** **seekoff()** **seekpos()** **set_user_data()** **stow()** **sync()**
bsamstr.h	**bsam_exit_list**	**remove()** **use()**
fstream.h	**filebuf** **(streambuf)**	**close()** **is_open()** **open()** **seekoff()** **seekpos()** **setbuf()** **sync()**
iostream.h	**streambuf**	**in_avail()** **out_waiting()** **sbumpc()** **seekoff()** **seekpos()** **setbuf()** **sgetc()** **sgetn()** **snextc()** **sputbackc()** **sputc()** **sputn()** **stossc()** **sync()**

Header File	Class (Base Classes)	Functions
stdiostream.h	**stdiobuf (streambuf)**	**is_open()** **seekoff()** **seekpos()** **stdiofile()** **sync()**
strstream.h	**strstreambuf (streambuf)**	**freeze()** **seekoff()** **seekpos()** **setbuf()** **str()** **sync()**

APPENDIX 3

Templates

Introduction

In C++, it is common for the same abstract data structure to be applied to different data types. For example, you could have a stack of integers, characters, or pointers. Rather than create a new stack class for each type, a template allows you to write generic type-independent code. A C++ template defines a family of types or functions by creating a parameterized type. Parameterized types can be used wherever actual types can be used. However, the template itself is not a type.

Templates are part of the evolving C++ language. The framework of the C++ language was created primarily by Bjarne Stroustrup. His book, *The C++ Programming Language, 2nd Edition*, provides good basic reference information but has limited coverage of templates.

Committee X3J16 has been working on an official standard for C++ for several years. During this time, the committee has standardized the existing features and added new features. Different implementations of the C++ language support different sets of features, exhibit different limitations, or exhibit different behavior in parts of the language that were not previously well defined.

Release 6.50 of the SAS/C C++ Development System supports templates as described in Stroustrup's book *The C++ Programming Language, Second Edition*, but implements clarifications to the specification introduced by the ANSI draft C++ standard, as well as a few language extensions. This appendix explains and clarifies the implementation of templates as provided by Release 6.50 of the SAS/C C++ Development System. The descriptions assume that you are already familiar with C++ templates.

Template Parameters

The SAS/C C++ Development System supports nontype template parameters as described in the ANSI draft C++ standard. Nontype template parameters types must be

- □ integer types
- □ enumeration types
- □ pointers to functions or objects
- □ references to functions or objects
- □ data member pointers.

Nontype template parameters can use previously declared template parameters in their declaration type. However, template parameters cannot be used in subexpressions within the type. For example:

```
template <class T, T* someObject>
class C { . . . };
int p;
C <int, &p> c;
template <int I, int a[I]* pArray> class C2; // ok
template <int I,
          int a[I*100]* pBigArray>  // error because
                                    // of subexpression I*100
class C2;
```

Template Arguments

Template arguments in the SAS/C C++ Development System follow these guidelines:

- □ Template arguments cannot be unnamed types or local types. Local types are declared inside a function.
- □ Nontype template arguments must be one of the following:
  - □ integral constant expressions (including **`enum`** constants)

- pointers to objects with external linkage (including static class members) specified as:
  &external_name or
  &class_name::static_member
- pointers to functions with external linkage specified as:
  external_name or
  class_name::static_member or
  &external_name or
  &class_name::static_member
- references to functions or objects with external linkage (including static class members) specified as:
  external_name or
  class_name::static_member
- member pointer constants for nonstatic data members specified as:
  &class_name::member

The C++ translator performs a limited number of conversions on explicitly specified template arguments in order to match the type of the formal template. The outer level qualifiers on the formal type are ignored for matching purposes. The standard conversions performed on template arguments are

- integral conversions
- reference binding where an object or function name argument of type **T** converts to a parameter of type **T&**
- function to pointer conversion
- qualification conversions

Template Declarations and Definitions

The syntax for declaring a class template is

```
template <class T> class C;  // T is a type parameter
```

Within the body of a class template or class template member definition, the class template name without a template argument list may be used as a synonym for the particular class specialization being instantiated. A specialization of a class template is referred to as a template class. For example:

```
template <class T> class C {
        C<T>* p1;
        C* p2;   // type is also C<T>*
        };
```

The SAS/C C++ Development System also supports the ANSI/ISO **typename** keyword. **typename** can be used as a substitute for the keyword **class** when declaring template type arguments, as follows:

```
template <typename T> class C;   // an alternate
                                 // declaration
```

Using Typename with Dependent Qualified Names

The primary use of **typename** is as a name modifier in template declarations. Dependent qualified names are defined as those names in which the qualifier scope explicitly depends on a template parameter that is part of the name. The **typename** keyword is required to precede dependent qualified names. This is necessary because declaration and expression parsing must be able to distinguish between type and nontype names before the types and values of the template arguments are known. Dependent qualified names cannot be resolved by lookup as they would be for a normal class of names.

For example, if **T** is a template parameter and **A** is a template class that has a single type parameter, then the names are referred to as dependent qualified names because the scopes of the names are dependent:

```
T::n
A<T>::n
```

In this example, the scope of the name is not dependent:

```
A<int>::n
```

An example of a template declaration that uses **typename** follows:

```
template <class T> class Vector {
        . . .
public:
   typedef T* Iterator;
   friend Iterator first( Vector<T>& );
   };
template <class T>
   typename Vector<T>::Iterator first( Vector<T>& v )
   { . . . };
```

The current release of the SAS/C C++ Development System only checks for **typename** when parsing declaration specifiers in a template declaration. However, the C++ draft requires the **typename** modifier on qualified dependent type names in the body of a template definition. You should consider using the **typename** keyword in new template code. For use with older compilers that do not support the **typename** keyword, the **typename** keyword can be hidden with the preprocessor directive as follows:

```
#define typename /* nothing */
```

Declaration Rules

Template classes and nonmember function templates may be declared multiple times in a translation unit, but they are subject to the single definition rule. The single definition rule states that a template may be defined at most once within each translation unit. However, a template may be defined in multiple translation units. Each specialization of a function or template class member must be defined in at most one translation unit.

Template class member declarations that are not friend declarations must be definitions.

Note: Template friend declarations of template class members are not supported in this release of the SAS/C C++ Development System. △

Template declarations may be used to declare friend classes and functions. However, friend template declarations may not be definitions.

For each template declaration, the template parameters can be specified with different names just as different function parameter names may be specified in different function declarations. For example:

```
// forward declaration of template C
template <class T, class U> class C;

// redeclaration of template C
template <class X, class Y> class C;
```

When declaring a template class member, the order of template parameters in the template class member must match the order of the corresponding parameters in the class template. The same order must also be used to specify the template parameters in the scope portion of the qualified member name being declared.

```
template <class T, class U>
class C {
public:
   static int i;
   static int j;
   };

template <class T, class U>
int C<T, U>::i = 0;   // OK

template <class T, class U>
int C<U, T>::j = 0;  // error - parameter order not
                     // the same
```

Note: In Release 6.50, template parameters may not have default arguments. △

Function Templates

The C++ language allows function templates in addition to class templates. Function templates provide the ability to create generic functions that are parameterized on types and constant values. The terms function template and template function are sometimes used interchangeably. Strictly, however, a function template is a declaration that specifies a family of overloaded functions. The declaration of a member function in a class template also is considered a function template. An instance of the family of overloaded functions associated with a function template is generated by specializing the template, which involves filling in particular types and values for the template parameters. A template function, then, is just an instance of a function template where particular values for all the template parameters have been determined.

The SAS/C C++ Development System supports function templates with nontype parameters. Refer to "Template Parameters" on page 117 for a description of nontype parameters. You can use nontype parameters to declare the template parameters and return types in a function template declaration. An example follows:

```
template <int SZ> class Bitvect;

template <int SZ1, int SZ2>
Bitvect<SZ1+SZ2> concat( const Bitvect<SZ1>& b1,
                         const Bitvect<SZ1>& b2 );
```

Nonmember function templates cannot have default arguments.

Overloading Function Templates

Function templates and ordinary nontemplate functions having the same name are not related. However, C++ allows for a function template to be overloaded with other function templates or nontemplate functions. During call overload resolution, each function template corresponding to the called name creates at most one candidate specialization of the template. The specialization is created by determining types and values for the template parameters through template argument deduction.

Specializations of a function template participate like ordinary functions in the overload resolution process. Once all functions for which there are better matches are eliminated, only the nontemplate functions that are left over are considered. Therefore, a function template is chosen only if it is a better match than any of the nontemplate functions.

Deducing Arguments

The template arguments for a member function of a template class are the actual template arguments for the template class. Otherwise, the template arguments are determined by deduction from the function parameter types.

Template argument deduction determines template parameter values by comparing each function parameter of the function template with the corresponding arguments of the call. Arguments that are not dependent on unspecified template parameters are matched to parameters exactly as they would be in normal function arguments. Otherwise, types and values are chosen for the template parameters which, when substituted in each function template parameter type, will match the corresponding argument type after trivial argument conversions. The trivial conversions performed on template arguments are

□ reference binding

Reference type arguments are dereferenced normally before type matching occurs. If a parameter type is a reference, the dereferenced type is used for matching purposes. For example:

```
template <class T> void f( T& );

void testit( int i )
{
   f( i );     // calls f( int& )
}
```

□ function to pointer conversion and array to pointer conversion

For arguments to nonreference parameters, the normal function to pointer and array to pointer conversions are applied before matching. For example:

```
template <class T> void f( T );

int a[5];
void testit()
{
   f( a );     // calls f( int* )
}
```

□ qualifier changes
Trivial qualifier changes are allowed. Pointer qualifier changes are allowed also. Outer level qualifiers are ignored for deduction purposes. For example:

```
template <class T> void f( const T* );

void testit( int* p )
{
     f( p );     // calls f( const int* )
}
```

In certain cases, template parameters cannot be deduced. For example, a template parameter used in the scope of a qualified name cannot be deduced:

```
template <class T>
void f( typename T::Z* ); // T not deducible
```

Template parameters used in subexpressions in a template function declaration cannot be deduced either. However, parameters that are used directly as array bounds and class template arguments are deducible. Several examples follow:

```
template <int sz>
void f1( int (*arrayPtr)[sz] );     // deducible

template <int sz>
void f2( int (*arrayPtr)[sz+1] ); // sz not
                                  // deducible

template <class T, int SZ> class Vect;

template <class T, int sz>
void f3( Vect<T, sz>& v );    // deducible

template <class T, int sz>
void f4( Vect<T, sz+1>& v );    // sz not deducible

template <class T>
void f5( Vect<T, 8/sizeof(T) >& v ); // T not
                                     // deducible

template <class T, int sz>
void f6( Vect<T*, sz >& v );  // deducible
```

Note: The major array bound of an array parameter to a function is ignored when deducing the parameter type:

```
template <int size>
void f7( int a[size][4] ); // size not
                           // deducible,
              // same as "int (*a)[4]"
```

△

Because the type of a function template that uses template parameters in subexpressions is not readily available, any redeclaration of the function template must match textually, at the token level, with the exception of template parameters which can be renamed. For example:

```
template <int N>
void f8( int(*ap) [N+1]);

template <int U>
void f8( int(*ap) [U+1]); // redeclaration -
                // template parameters renamed
```

Specifying Arguments

Template arguments for nonmember template functions may be specified in the form:

```
function_template_name< optional_template_args >;
```

The function template name is followed by a possibly empty (**< >**) template argument list.

Explicit template argument list specifications for function templates are an ANSI/ISO extension. They are used as a function name in expressions and in explicit specialization and instantiation declarations when the template has parameters that cannot be deduced from the context. For example, if a function template has template parameters that are not deducible, then an explicit template argument list containing actual values for those parameters must be used to call it.

Effectively, explicit template arguments are used to select a subset of the function templates that match the specified arguments. Then the parameters that correspond to the specified arguments are fixed. For example:

```
template <class T, class U>
inline T implicit_cast( U u ) { return u; }
```

This example produces a function that will perform an implicit conversion to the specified type, therefore avoiding a cast that could allow unsafe conversions:

```
void f( double );
void f( int );

template <class T>
void callit( T t )
{
   // call f( double )
   f( implicit_cast<double>( t ) );
}
```

Not all the arguments for the function template need to be specified when using an explicit template argument list. The specified arguments are substituted for the corresponding parameters in the declared type of each matching template. Any template arguments not specified are deduced in the normal manner. Function arguments for which all template parameters are specified are treated like normal function arguments for overload resolution purposes. For example:

```
template <class T>
inline T Max( T x, T y )
 { return (x > y ) ? x : y; }

int compare( int i, char c )
{
    // Calling Max( i, c ) would fail because
    // the argument types differ too much,
    // and the template parameter cannot
    // be deduced.  But you can explicitly
    // specify the type.
    Max<int>( i, c );  // OK, calls Max(int, int)
}
```

As an alternative to explicit template argument specification, older C++ implementations allow a construct called a guiding declaration. A guiding declaration uses an ordinary declaration to specify a specialization of a function template. A guiding declaration is used to avoid the same deduction problem illustrated in the previous example.

```
template <class T>
inline T Max( T x, T y )
 { return (x > y ) ? x : y; }

int Max( int, int );  // guiding declaration

int compare( int i, char c )
{
   // Calling Max( i, c ) would fail without
   // the guiding declaration because the
   // argument types differ too much.
   Max( i, c );  // OK, calls Max(int, int)
}
```

When the **tmplfunc** translator option is specified, nontemplate functions are distinct from template specializations. Therefore, guiding declarations are not recognized. In such cases, inline functions can be used in place of the guiding declaration:

```
template <class T>
inline T Max( T x, T y )
 { return (x > y ) ? x : y; }

// forwarding declaration
inline int Max( int x, int y )
{  return Max<>( x, y ); }  // call the template
                            // version

int compare( int i, char c )
{
    // Calling Max( i, c ) would fail without
    // the forwarding declaration because the
    // argument types differ too much.
    Max( i, c );  // OK, calls Max(int, int)    }
```

See "Option Descriptions" on page 38 for a full description of the **tmplfunc** option.

Function Template Signatures

Functions with normal linkage conventions (not **extern "C"**) encode information about their parameter types into their linkage name. For specializations of function templates, this linkage information potentially includes the template arguments also. When the template arguments are included, components such as COOL will print the name including the template arguments using the function name followed by the actual template arguments.

The ANSI/ISO C++ draft treats specializations of function templates as distinct objects from non-template declarations with the same name and type. Effectively, the linkage names of template specializations are different from non-template linkage names. Older C++ compilers treat non-template declarations that match template specializations as the same object. Therefore, the same linkage name is used.

By default, the SAS/C C++ Development System uses the nontemplate linkage name for specializations of normally deducible function templates for compatibility with older code. Functions which are not deducible (see message LSCT628 in *SAS/C Software Diagnostic Messages, Release 6.50*), which older compilers treat as erroneous, always include the template parameters in their linkage. With the **tmplfunc** option, the linkage names of function template specializations will be made distinct from nontemplate functions, by including the template arguments for the specialization in the linkage name.

Other Template Signatures

When the translator or other tools refer to names with C++ linkage whose scope or type depends on templates, the name printed out contains template argument lists. For the most part, the interpretation of these template arguments should be obvious. These tools have incomplete information about the types and definitions involved, so some special cases are worth noting:

- Template arguments are not output in tracebacks. In tracebacks, templates are denoted with an empty argument list **< >**. Also the argument list may be omitted in cases where there is not enough space to output the list completely.
- Enumeration constants are described by their numeric value.

  ```
  enum Color { Red = 1, Blue = 2, Green = 4 };
  template <Color clr> class C
  {
  public:
      static int si;
  };
  ```

  A reference to **C<Red>::si** might be output as **C<1>::si**.
- Data member pointers are printed in terms of their offset within the structure.

In the latter two cases, there may be no unique C++ expression that generates the given value, so a standard form was chosen.

Template Specialization Declarations

An *explicit specialization* is a declaration of a specialization of a template or template member that overrides generation of the specialization from its template definition. Such a specialization is called explicitly specialized. Members of explicitly specialized classes are treated as if they too are explicitly specialized.

Explicitly specialized items must be declared before their first use in every translation unit in which they are used. If the specialized item is used, it must be defined in a translation unit within the program.

The ANSI/ISO C++ draft form for an explicit specialization is a global declaration of the form:

```
template < > DECLARATION
```

The declaration refers to a specialization of a previously declared template or template member. The declaration may be a definition. If the declarator names a function template, an explicit template argument list may optionally follow the template name in order to disambiguate the specialization being declared. For example:

```
template <class T> class TypeAttr {
public:
   static const int isIntegral;
   typedef T* Pointer;
   };

template <class T>
const int TypeAttr<T>::isIntegral = 0; // default to
                                       // non-integral

template < >
const int TypeAttr<int>::isIntegral = 1;

template < >
const int TypeAttr<long>::isIntegral = 1;

template <class T>
T& deref( typename TypeAttr<T>::Pointer ); // T not
                                           // deducible

template < >
int& deref<int>( int *p )  // have to specify <int>
                           // to disambiguate
{ return *p }
```

For compatibility with older compilers, if the **TMPLFUNC** option is not specified, ordinary function declarations that match the types of template specializations are treated as explicit specializations. As a special case, for compatibilty with older compilers, friend declarations inside a template class that use template parameter names in the declaration type are not treated as specializations. An example follows:

```
#include <iostream.h>

template <class T>
class Vector {
    // the following uses template
    // parameter "T", not a specialization
    friend ostream& operator<<
          ( ostream&, Vector<T>& );

    // the following does not use
    // template parameters
    // It is treated as an explicit specialization
    friend ostream& operator<<
          ( ostream&, Vector<int>& );
    . . .
    };

// define the template
template <class T>
ostream& operator<<( ostream& o, Vector<T>& v )
{ . . . }

// the following definition is also
// treated as an explicit specialization
ostream& operator<<( ostream& o, Vector<double>& v )
{ . . . }
```

Template Instantiation

Release 6.50 of SAS/C C++ supports both explicit instantiation and automatic instantiation of templates. The following two sections describe their features.

Explicit Instantiation

An explicit instantiation is a declaration with global scope of the form:

```
template declaration
```

where *declaration* specifies a template class member, a template function, or a template class name, for example,

```
class C<int>
```

An explicit instantiation must be a global declaration.

The template specialization declared is generated from its template definition in the current translation unit, even if the specialization is not used. If the **autoinst** option is not used, a nonstatic noninline template function or static data member of a template class may only be explicitly instantiated in one translation unit.

When declaring a specialization of a function template, a template argument list optionally may follow the function template name in order to disambiguate the declaration.

```
template <class X>
class C
{
   static X* sp;
};
```

```
template <class T>
T* C<T>::sp = 0;

template C<int>::sp;  // declare C<int>::sp here
```

The declaration may not be a definition. The template definition must be available at the point of the explict instantiation declaration.

Automatic Instantiation

The bodies of template classes and inline (or static) template functions are always instantiated implicitly when their definitions are needed. Member functions of template classes are not instantiated until they are used. Other template items can be instantiated by using explicit instantiation. However, explicit instantiation becomes unwieldy with more complex uses of templates. When the **autoinst** option is used, nonstatic noninline template functions and static data members of template classes are automatically implicitly instantiated. The restrictions on automatic instantiation are described below.

Implicit instantiation of a template item with **autoinst** requires that the option be enabled on a compilation unit, which contains both a use of the item and its corresponding template. Often, this can be arranged by including the definition of any needed templates in the header file which declares the templates.

With the **autoinst** option, the compiler organizes the output object module so that COOL can arrange for only one copy of each template item to be included in the final program. The original compilation unit from which the final copy is generated and the section name created for the template item are unpredictable, although the section name can be determined from the COOL listing.

Each nonstatic noninline template function or static data member of a template class that is used must be either implicitly instantiated by **autoinst**, explicitly instantiated in exactly one compilation unit, or else explicitly specialized and defined. Otherwise, COOL will diagnose a missing definition for the template item.

The draft C++ standard requires that nonstatic templates and inline functions that are defined in more that one compilation unit refer to the same set of objects and functions with each definition. Implicit instantiation with the **autoinst** option depends on this requirement being satisfied. Nonstatic template definitions of functions and static data members may not refer to global static objects and functions if the definition appears in more than one compilation unit. Uses of global statics inside a template item instantiated from such definitions will be diagnosed with **WARNING 683: Implicit template instantiation uses static object**. For more information on this warning message, see *SAS/C Software Diagnostic Messages.* An example follows:

```
static int i;

template <class T>
void collect( const T& t )
{
   i += t;   // use of static 'i' gets warning 683
}

void testit()
{
   collect( 1 );  // cause instantiation of
                  //collect( const int& )
}
```

Because global static objects and functions are available only in their original compilation units, use of statics in items implicitly instantiated from templates force the item to be inserted in the current compilation unit as if the template item were explicitly specialized. This insertion is diagnosed with **WARNING 684: Static use forces automatic template instantiation in primary module**. For more information on this warning message, see *SAS/C Software Diagnostic Messages.* This insertion may cause a multiple definition error for the template item at link time if the template was defined in more than one compilation unit.

Calls from template definitions to inline functions that use global static objects or functions cause similar problems in the direct use of those statics. Therefore, **WARNING 685: Inline function uses static object** will be generated for use in nonstatic inline functions. The function will be treated as static for the purposes of warning messages 683, 684, and 685. Warnings 683 and 684 are only output for items automatically instantiated by **autoinst** from template definitions. The definitions of explicitly specialized template items may reference global statics without generating warnings 683 or 684. For more information on these warning messages, see *SAS/C Software Diagnostic Messages.*

APPENDIX 4

Pointer Qualification Conversions, Casts, and Run-Time Type Identification

Pointer Qualification Conversions

Compilers for ANSI C and older C++ compilers allow implicit pointer qualification conversions such as **`int**`** to **`const int**`** that are not type-safe. For example:

```
void set_it( const int** target, const int* source )
{
   *target = source;
}

void test_it( const int* const_ptr )
{
   int* nonconst_ptr;

   // Get a non-const copy of const_ptr
   // without casts!!
   // The first argument uses the old unsafe
   // implicit conversion.
   set_it( &nonconst_ptr,
           const_ptr );

   *nonconst_ptr = 0; // assignment through
                      // const pointer
}
```

To avoid a similar problem with the C++ type system, the draft version of C++ allows only a subset of the implicit qualification conversions allowed in ANSI C. This subset can be summarized as follows:

- Conversion of single level pointers work as they did in ANSI C.
- With multiple pointer levels, when qualifiers change on a pointed to type, intermediate pointed to types in the target type must be **`const`**. For example, an **`int**`** pointer cannot be implicitly converted to **`const int**`** but it can be implicitly converted to

```
const int* const *
```

For more information on implicit pointer qualification conversions, refer to section 4.4 of the ISO C++ draft.

Cast Operators

This release of the C++ translator supports several new cast operators. These operators are defined in the ISO C++ draft as improved alternatives to the old cast operators, for example, **`(type)(expr)`** or **`type_name (expr)`**. For more information on these operators, see Stroustrup's *Design and Evolution of C++*.

The new cast operators are

`static_cast< type > ( expr )`
portable conversions that do not cast away constness

`const_cast< type > ( expr )`
pointer qualification conversions, including the unsafe conversions which cannot be performed implicitly

`reinterpret_cast< type > ( expr )`
nonportable conversions such as:

- conversion from pointer to int and back
- conversions between arbitrary pointers to objects
- conversions between arbitrary pointers to functions

`dynamic_cast< type > ( expr )`
run-time type dependent pointer and reference conversions.

The new cast operators, except **`dynamic_cast`**, provide separate components of the functionality of the old casts that allow improved type checking. Any old-style cast can be expressed as a combination of the new casts. The new cast operators, other than **`const_cast`**, are not allowed to cast away constness on pointer, pointer to member, and reference conversions. See section 5.2.11 of the ISO draft for more information.

Run–Time Type Identification Requirements

Release 6.50 of the SAS/C C++ Development System supports Run–Time Type Identification, or RTTI. RTTI enables the compiler to automatically generate type information for objects checked at run time. The **`RTTI`** option must be specified for each compilation unit to assure that class objects constructed with functions defined in the unit have the information required for dynamic type identification by the **dynamic_cast** and **typeid()** operators. See "Option Descriptions" on page 38 for more information on the **RTTI** option. The generated code will abort if the **dynamic_cast** or **typeid()** operators are applied to C++ class objects with virtual functions which do not have RTTI information. This means it is generally unsafe to have a program that uses **dynamic_cast** or **typeid()** but does not generate RTTI information in all of its compilation units.

Compilation units that do not use **dynamic_cast** or **typeid()** can be compiled with the RTTI information. The resulting object files can be safely linked into progams that do not use RTTI. The C++ library is compiled this way. Note that dynamic type identification applies only to C++ classes with virtual functions, so there is no compatibility issue with non-C++ code.

ISO C++ specifies that certain erroneous uses of **dynamic_cast** and **typeid()** cause a C++ exception to be thrown. Release 6.50 of the SAS/C C++ Development System does not support exceptions, so a mechanism corresponding to default handling of an exception, with no user handler, is invoked for these errors. Effectively, the **terminate()** operator is called, which aborts the program by default. However, a handler can be specified by **set_terminate()** to perform cleanup before terminating program execution. See "typeinfo.h Header File" on page 48 for more information on these operators.

As an extension to ISO C++, the run-time code for RTTI will detect when dynamic type information is requested for an object that was not compiled with the **RTTI** option. In such cases, the **terminate()** mechanism is used to abort execution.

The **typeid()** operator returns a reference to a statically allocated object. The destructor should never be called for this object. Calling **typeid()** for the same type in different compilation units may produce references to different objects. Always use operator **==** or operator **!=** to test for type equality.

Index

Your Turn

If you have comments or suggestions about *SAS/C C++ Development System User's Guide, Release 6.50*, please send them to us on a photocopy of this page or send us electronic mail.

For comments about this book, please return the photocopy to

SAS Institute Inc.
Publications Division
SAS Campus Drive
Cary, NC 27513
email: yourturn@unx.sas.com

For suggestions about the software, please return the photocopy to

SAS Institute Inc.
Technical Support Division
SAS Campus Drive
Cary, NC 27513
email: suggest@unx.sas.com