M A C I N T O S H
P R O G R A M M E R ' S
W O R K S H O P

SC / SCpp

C/C++ Compiler for 68K Macintosh

Version 1.0

030-9409-A
Developer Press

Apple Computer, Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

LIMITED WARRANTY ON MEDIA AND REPLACEMENT

If you discover physical defects in the manual or in the media on which a software product is distributed, APDA will replace the media or manual at no charge to you provided you return the item to be replaced with proof of purchase to APDA.

ALL IMPLIED WARRANTIES ON THIS MANUAL, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF THE ORIGINAL RETAIL PURCHASE OF THIS PRODUCT.

Even though Apple has reviewed this manual, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS SOLD "AS IS," AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS MANUAL, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Tables and Listings

About This Book

This book is a reference for SC and SCpp, which are ANSI-compliant C and C++ compilers that produce optimized code for the Macintosh environment. It describes the use and features of the SC and SCpp compilers.

The SC and SCpp compilers run within MPW on both the Power Macintosh and 68K families of computers, but generate code only for the classic 68K and CFM-68K runtime environments. The compilers are part of a tool suite for the Mac OS environment that includes a debugger, an assembler, and a linker. You can use SC and SCpp to compile 68K source code for the 68K classic and CFM-68K runtime environments. You can also use them in combination with a PowerPC™ compiler (such as MrC and MrCpp) and related tools to create applications that contain both PowerPC and 68K code. You cannot use SC or SCpp to generate PowerPC code.

To use this book, you need to be familiar with

- the MPW development environment
- the C and C++ languages
- developer tools (compilers, linkers, and debuggers)
- standard compiler concepts (compilation phases, optimizations, and libraries)

This book contains the following three chapters:

- Chapter 1, “About SC,” provides an overview of the features of SC and SCpp, including the language dialects and extensions they support and certain implementation-defined details.
- Chapter 2, “Using SC,” explains how to compile C and C++ source files, properly handle the files used during compilation, choose the alignment convention for data structures, interpret error and warning messages, and set the compiler optimization level.
- Chapter 3, “SC Reference,” provides a detailed description of each compiler option and pragma and describes the predefined symbols.

Related Documentation

For information on using MPW, see the books *Introduction to MPW*, *Building and Managing Programs in MPW*, and *MPW Command Reference*. Where appropriate, this book points you to the pertinent parts of the referenced books.

For information on developing Macintosh applications, see the *Inside Macintosh* documentation suite. If you have not previously developed Macintosh applications, or are not familiar with the *Inside Macintosh* documentation suite, begin with *Inside Macintosh: Overview*.

Note that this book does not describe in detail the contents of the ANSI C library (StdCLib) and the C++ library (CPlusLib). There are a number of books available that describe the standard C and C++ libraries. There are also books on the C and C++ languages, such as *The American National Standard Institute—Programming Language—C*, commonly referred to as "the ANSI standard," and *The Annotated C++ Reference Manual*. For further information, visit your local bookstore. For a description of the Apple extensions to the ANSI C library, see *Building and Managing Programs in MPW*.

Conventions Used in This Book

This book uses various conventions to present information. Words that require special treatment appear in specific fonts or font styles. Certain information, such as command line options, uses special formats so that you can scan it quickly.

Special Fonts

This book uses several typographical conventions.

All code listings, reserved words, names of actual data structures, constants, fields, parameters, and routines are shown in Letter Gothic (`this is Letter Gothic`).

Words that appear in **boldface** are key terms or concepts and are defined in the glossary.

Command Syntax

This book uses the following notations to describe compiler commands:

`[ ]`	Brackets indicate that the enclosed elements are optional.
`a | b`	A vertical bar indicates an either/or choice.
italic	Italic font indicates that you must substitute a real value that matches the definition of the element.
. . .	Ellipses (. . .) indicate that the preceding item can be repeated one or more times.

Types of Notes

This book uses two types of notes.

Note
A note like this contains information that is useful but not essential for an understanding of the main text. ◆

IMPORTANT
A note like this contains information that is crucial to understanding the main text. ▲

For More Information

The *Apple Developer Catalog* (ADC) is Apple Computer's worldwide source for hundreds of development tools, technical resources, training products, and information for anyone interested in developing applications on Apple computer platforms. Customers receive the *Apple Developer Catalog* featuring all current versions of Apple development tools and the most popular third-party development tools. ADC offers convenient payment and shipping options, including site licensing.

To order products or to request a complimentary copy of the *Apple Developer Catalog*, contact

Apple Developer Catalog
Apple Computer, Inc.
P.O. Box 319
Buffalo, NY 14207-0319

Telephone	1-800-282-2732 (United States) 1-800-637-0029 (Canada) 716-871-6555 (International)
Fax	716-871-6511
AppleLink	ORDER.ADC
Internet	order.adc@applelink.apple.com

If you provide commercial products and services, call 408-974-4897 for information on the developer support programs available from Apple.

For information about registering signatures, file types, Apple events, and other technical information, contact

Macintosh Developer Technical Support
Apple Computer, Inc.
1 Infinite Loop, M/S 303-2T
Cupertino, CA 95014

CHAPTER 1

About SC

Contents

This chapter provides a general introduction to the SC and SCpp compilers. These are C and C++ compilers, respectively, and they run as MPW tools. In this book, *SC* is used to refer to both compilers unless there is a reason to distinguish between them.

The SC Compiler

The SC compiler runs on the 68K Macintosh computers and natively on Power Macintosh computers and produces code for the classic 68K and CFM-68K runtime architectures. The SC compiler is distributed with standard C and C++ header files and libraries as well as sample code. You handle all coding tasks, such as editing source files and entering commands, using standard MPW methods.

You cannot generate PowerPC™ code with SC. You can, however, create fat applications using SC together with a PowerPC compiler (such as MrC or MrCpp) and related tools, such as the MPW resource compiler. A **fat application** is one that contains both PowerPC and 68K code. See *Building and Managing Programs in MPW* for more information.

The computer you use to compile the code (the host computer) can be either a 68K or Power Macintosh computer. The minimum system configuration for the host computer is a Macintosh II computer with 20 MB of free hard disk space, 20 MB of RAM, system software version 7.0.1 or later, and MPW 3.3 or later. For an optimal development environment, however, Apple recommends a Power Macintosh computer. In either case, it is best to provide an initial minimum MPW partition size of 8 MB for the SC compiler. You may also need to increase the MPW Shell stack size, using the `SetShellSize` command.

The minimum configuration for your **target computer,** or the computer on which you will execute the compiled code, is a 68K Macintosh computer. The system software on the target computer must be compatible with the header files and libraries used to build your application on the host Macintosh computer.

Language Support

A **source code dialect** is a specific version of a programming language. The SC compiler accepts source code files that adhere to one of two source code dialects:

- The **ANSI C language dialect** adheres to the ANSI C standard. The ANSI Language dialect is the language defined by the American National Standards Institute (ANSI) in the document *American National Standard Institute* (ANSI X3.159-1991).
- The **C++ language dialect** adheres to the de facto C++ standard except for templates and exception handling. The **de facto C++ standard** is the ANSI working paper *American National Standard Institute* (ANSI X3J16), which is based on CFront version 3.0 from USL (UNIX® System Laboratories).

You use the `SC` command to compile C source code files and the `SCpp` command to compile C++ source code files. Because the compiler does not depend on the filename to identify the source code dialect, you can name your source files as you wish. However, there are some general conventions for C and C++ source filenames that may be required by other development tools in the MPW environment. The typical filename suffix for a C++ file is `.cp`, whereas `.c` indicates that the file is in the ANSI C dialect.

Strict C and C++ Language Standards

Unless you request it, SC and SCpp do not conform strictly to the ANSI C and C++ language standards when compiling. This means that, by default, SC issues warning messages rather than error messages for certain nonconforming constructs in your source code file and attempts compatibility with CFront 3.0 for the C++ dialect. You can use the `-ansi on` or `-ansi strict` options to specify a strict conformance to the C or C++ language definition: `-ansi on` enforces ANSI standards but allows you to use Apple extensions (see “Apple Language Extensions” (page 1-6)); `-ansi strict` does not allow you to use Apple extensions. In either case, SC issues error messages rather than warning messages for any nonconforming use. For brevity, this book uses the terms **strict ANSI C** and **strict C++** to indicate use of the stricter language definitions. (You can read more about error and warning messages in “SC Messages” (page 2-12).)

Most of the header files supplied with SC require using the `-ansi off` option (the default setting for the SC compiler), except for those header files defined in the C and C++ language standards.

IMPORTANT

The order of expression evaluation is not guaranteed with the SC compiler beyond that specified by the ANSI C standard. For example, there is no restriction on the order in which parameters are computed for a function call. It is therefore never safe to write code that depends upon an order of evaluation or that depends upon operator side effects occurring at any particular point in the evaluation of an expression, other than that specified by the ANSI C standard. ▲

Here are some restrictions to bear in mind when using the strict ANSI standard for either C or C++:

- You cannot use arithmetic on pointers to functions.
- You cannot use the `sizeof` operator to get the size of a function.
- You cannot use binary numbers.

In addition, there are C-specific restrictions:

- You cannot use C++-style comments in C code.
- You cannot have an empty struct or union definition.

There are also C++-specific restrictions:

- Anonymous unions must be static.
- Member functions cannot be static.
- You cannot generate a reference to a temporary variable.
- You cannot convert to and from the `void` data type.

A complete list of restrictions is provided in the appendix.

Implementation-Specific Details

In addition to choosing whether or not to use the strict ANSI C standard, you can set **implementation-specific details,** which are compilation or language features that the ANSI C language definition leaves to the discretion of the implementor. For example, you can use the `-enum` option to determine the size used for the `enum` type in your code, or you can use the `-typecheck` option to relax type checking. See “Language Options” (page 3-7), for more information about language-specific options.

Libraries

Refer to the *MPW Standard C Library Reference* for libraries that support the classic 68K and the CFM-68K Macintosh architecture. These include Apple Extensions, Numerics, and the CFM-68K runtime libraries.

Apple Language Extensions

By default, SC accepts source code that includes three Apple language extensions:

- the `pascal` keyword, to indicate that a function follows Pascal calling conventions
- Pascal strings (a length byte followed by the string characters)
- C++-style comments in C code

The Pascal Keyword

You can use the `pascal` keyword to declare Pascal calling conventions for a function or function pointer. With the classic 68K runtime architecture, the Pascal calling conventions differ from the C and C++ calling conventions. With the PowerPC and CFM-68K runtime architectures, however, these calling conventions are identical. If you are compiling for CFM-68K, the compiler recognizes the `pascal` keyword but ignores it. You should use the `pascal` keyword in your source code only where it is syntactically legal to use the

`const` or `volatile` keywords. The `pascal` keyword applies only to function types. You can declare a pointer to a Pascal function, however, by declaring a pointer to a function type that has been modified by the `pascal` keyword. For instance, consider these declarations:

```
pascal void SomePascalFunction(some_type b) { }
pascal void (* pfptr)(some_type a);
typedef pascal void PascalFunctionType(int);
PascalFunctionType * arg;

pascal void SomeOtherPascalFunction(pascal void (*)(int));
```

`SomePascalFunction` uses Pascal calling conventions. A call to the function pointed to by `pfptr` uses Pascal calling conventions. `PascalFunctionType` is the name of a type that returns `void`, has an argument of type `int`, and uses Pascal calling conventions. The argument `arg` is a pointer to a function of this type. `SomeOtherPascalFunction` uses Pascal calling conventions; its only parameter is a pointer to a function that returns `void`, has a single parameter type of type `int`, and also uses Pascal calling conventions. Note that it is necessary to apply the `pascal` keyword twice there.

You must consistently apply Pascal conventions to a given function in the same program. That is, if you declare Pascal conventions for one occurrence of a given function, you must also declare Pascal conventions for all other occurrences of that function. Consider these declarations:

```
class Bat {
    pascal void Fly(int speed);
};
class Bird {
    pascal void Fly(int speed);
};
```

The functions `Bat::Fly(int)` and `Bird::Fly(int)` do not conflict with each other. Both functions are Pascal functions. Inconsistent declarations can result in an error.

Do not declare a C++ operator function as a Pascal function. Even though the SC compiler does not distinguish Pascal functions from C functions, it does change the overload rules for C++. In the PowerPC runtime architecture, Pascal function names are mangled in the same way as C++ external functions. Pascal

member functions are handled in the same way as C++ member functions, which cannot be overloaded by parameter type. Here are some illegal declarations:

```
class Bat {
    pascal void Fly(int speed);
    pascal operator new(size_t size);
};
pascal void Beep(void);
pascal void Beep(int duration);
```

It is not legal to overload Pascal functions in this way. The function `Bat::operator new(size_t)` cannot be a Pascal function because it is an operator function. The second declaration of the `Beep` function is illegal because you cannot declare a function previously declared as something else.

Pascal Strings

The SC compiler allows you to use Pascal strings in your C or C++ source code. A Pascal string is a sequence of characters that begins with a length byte, uses `\p` as its first character (for example, "`\pfileName`"), and has a maximum size of 255 characters.

The `\p` is treated as one character and indicates a Pascal string literal. Pascal strings are stored differently than C strings. Both strings are arrays of characters whose last element is the null byte (`\0`). Pascal strings, however, have an initial length byte. For example, the Pascal string "`\pabc`" takes up 5 bytes: `\03`, `a`, `b`, `c`, and `\0`.

You can concatenate Pascal and C strings in any combination. The following example shows uses of Pascal and C strings:

```
"\p"                /* pascal string of length 0 */
"\pabc"
"\pabc" "def"       /* becomes "\pabcdef", a Pascal-style string */
"def" "\pabc"       /* becomes "defabc", a C-style string */
```

C++-Style Comments in C Code

SC allows you to place C++-style comments in your C source code. The ANSI standard defines C-style comment delimiters as `/*` and `*/`. The SC compiler ignores anything enclosed between the opening (`/*`) and closing (`*/`) delimiters. C++-style comments use the delimiter `//` at the beginning of the comment. The SC compiler ignores anything between the delimiter `//` and the end of the line.

The following example shows C++-style comments placed in C code:

```
void eatchar (void)
{
    int c;          // Here is a C++-style comment.
    c = getchar();
}
```

Although both comment styles are allowed, the examples in this book use only C-style comments.

Note
For C++, the ANSI standard allows C-style comments as well as C++-style comments. ◆

CHAPTER 2

Using SC

Contents

This chapter shows you how to use the SC compiler. It describes how to construct a compiler command line and summarizes the command line options. This chapter also provides some information about the diagnostic messages that the compiler can produce and describes the other types of compiler output you can generate. Finally, it shows how to choose the level of code optimization performed by the compiler.

This chapter is not a complete guide to building an application or preparing it for debugging. See *Building and Managing Programs in MPW* and *Macintosh Debugger Reference* for information on these subjects.

Compiling Files

After starting the MPW Shell application, you can enter commands in the Worksheet or any open document window. A command to run the SC compiler typically uses the following syntax, which includes the SC invocation, options, and source filename. (If you don't type in *filename* you will bring up a list of options.)

`SC` [*option list*] *filename*

Invoking SCpp is similar:

`SCpp` [*option list*] *filename*

You can place one source file and multiple options in a single SC command in any order. Neither filenames nor options are case sensitive. For example, to compile a C file, you might enter this command:

```
SC -o hello.o hello.c
```

Here, `hello.c` is your source file. The `-o` option specifies that the name of the resulting object file be `hello.o`. Using the SC command instead of the SCpp command indicates that you want to compile in ANSI C.

To compile a C++ file, use SCpp:

```
SCpp -o hello.o hello.cp
```

To prepare a source file for debugging with the Macintosh debugger for PowerPC, you must compile it using the `-sym on` option, which produces source position and symbol table information for the debugger. You enter the command in the following form:

```
SC -sym on -o hello.o hello.c
```

Note
The `-sym` option is not compatible with any SC optimizations and automatically suppresses their use. With `-sym on` there is no optimization. ◆

You might prefer to create an MPW makefile to execute your compiler commands. See *Building and Managing Programs in MPW* for information about using makefiles to automate the build process.

Table 2-1 summarizes the SC command line options. Chapter 3, "SC Reference" provides more detailed descriptions of each option, including definitions of the option parameter values. The page numbers in Table 2-1 refer you to the descriptions of each option in Chapter 3. For another source that describes the compiler options, use the `help SC` command in MPW.

Table 2-1 List of SC command line options

Option	Purpose
`-align` *parameter*	Sets alignment (page 3-7).
`-ansi` *parameter*	Enforces strict ANSI compatibility (page 3-14).
`-auto_import size`	Specifies that data over a certain size will reside in separate code fragments (page 3-27).
`-b` or `-b2` or `-b3` or `-b4`	Controls overlay strings, PC-relative strings, and functions (page 3-24).
`-bigseg`	Specifies that code is to reside in single segment fragments (page 3-26).
`-c`	Checks the syntax of your code without generating an object file (page 3-29).

continued

Table 2-1 List of SC command line options (continued)

Option	Purpose
`-char` *parameter*	Controls nature of default `char` data type (page 3-9).
`-d` or `-define`	Defines a preprocessor symbol name (page 3-15).
`-dump` *filename* or `-dumpc` *filename*	Saves the state of the compilation in a file (page 3-40).
`-e`	Generates preprocessor output (page 3-29).
`-elems881`	Generates inline with 68881 instructions (page 3-23).
`-enum` *parameter*	Controls or specifies the size of enums (page 3-11).
`-fatext`	Specifies that compilers should use target-unique extensions for output file names (page 3-42).
`-frames`	Generates stack frames (page 3-33).
`-i pathname[.pathname...]`	Sets header file path (page 3-42).
or	
`-I pathname[.pathname...]`	
`-inline` *parameter*	(C++ only) Controls the expansion of inline function within the code (page 3-20).
`-j0`	Recognizes 2-byte sequences for Japanese characters (page 3-12).
`-j1`	Recognizes 2-byte sequences for Taiwanese or Chinese characters in string and character constants (page 3-12).
`-j2`	Recognizes 2-byte sequences for Korean characters in string and character constants (page 3-13).
`-l` *filename*	Generates a source file listing (page 3-30).

continued

Table 2-1 List of SC command line options (continued)

Option	Purpose
`-load` *filename* or `-loadc` *filename*	Restores the saved compilation (page 3-41).
`-m`	Generates 32-bit references for data (page 3-31).
`-mbg`	Generates MacsBug names (page 3-35).
`-mc68020`	Generates MC68020 code (page 3-23).
`-mc68030`	Generates MC68030 code (page 3-23).
`-mc68040`	Generates MC68040 code (page 3-23).
`-mc68881`	Generates MC68881 code (page 3-23).
`-model`	Specifies memory model (page 3-25).
`-noMapCR`	Controls the interpretation of the newline `\n` and carriage-return `\r` characters (page 3-15).
`-nomfmem`	Specifies that the compiler should not use MultiFinder memory (page 3-35).
`-notOnce`	Allows SC to read multiple copies of header files (page 3-43).
`-o` *filename*	Sets output file (page 3-31).
`-onefrag`	Specifies that C++ constructs are in the same code fragment (page 3-26).
`-opt` *parameter[,modifier...]*	Optimizes code (page 3-21).
`-p`	Prints progress messages (page 3-32).
`-paslinkage`	Requests that, by default, the compiler use Pascal calling conventions (page 3-24).
`-proto` *parameter*	(C only) Generates prototypes (page 3-16).
`-s segname`	Assigns code to segment name (page 3-25).
`-sym` *parameter*`[`*,modifier...*`]`	Generates debugging information (page 3-34).

continued

Table 2-1 List of SC command line options (continued)

Option	Purpose
`-trace`	Generates function preambles and postambles to allow code profiling (page 3-36).
`-typecheck` *parameter*	(C only) Controls type checking (page 3-17).
`-u`	Suppresses predefined non-ANSI macros (page 3-27).
`-v`	Prints progress messages (page 3-36).
`-w` *parameter*	Controls warning messages (page 3-38).
`-x`	Allows more than the default maximum number of errors (four) without terminating the compilation (page 3-39).
`-xa` *parameter*	(C++ only) Specifies template access (page 3-19).
`-xi` *template*	(C++ only) Instantiates a template (page 3-18).

Handling Files

When you compile with SC, you generally work with three different types of files: source, object, and header.

- **Source files**: Text files that contain C or C++ source code.
- **Object files**: Files that contain relocatable machine code and result from compiling source files.
- **Header** (or **include**) **files**: Files that you merge into your source file using the `#include` preprocessor directive. These files often contain common variables and function declarations.

This section explains how to name files, how to include header files in your source code, and how to include the same header file more than once.

Using Proper Filename Conventions

When naming files, you need to be aware of certain name restrictions as well as the circumstances under which SC names files for you.

There are no restrictions for naming object and export list files. SC accepts any file prefix and extension that you assign to these files. For source files, however, the filename portion of a pathname cannot exceed 29 characters. When selecting your source file suffix, remember that although SC and SCpp allow any suffix, other tools require `.c` for C files and `.cp` for C++ files. See the section "Language Support" (page 1-4) for more information.

SC assigns filenames only if you do not name an object file on the SC command line. If you do not name an object file on the SC command line with the `-o` option, SC uses the source filename with a `.o` extension as the default object filename (for example, the source file `myfile.c` results in an object file named `myfile.c.o`).

Including Header Files

The format that you use to include a header file in your source file determines the search path order that the SC compiler uses to locate the header file. You can control how and where SC searches for header files by using quotation marks, angle brackets, or a colon.

- Quotation marks, or double quotes ("), enclosing a header filename indicate that SC should first search the directory where the source file is, then the directories named in the `-i` option, and finally, the directory pointed to by the Shell variable `{CIncludes}`, which is typically `{MPW}Interfaces:CIncludes`. Use double quotes to include your own header files.
- Angle brackets (< and >) enclosing a header filename indicate that SC should first search the directories named in the `-i` option and then the `{CIncludes}` directory. Use brackets to include system header files.
- A colon (:) within a header filename indicates an explicit pathname. SC does not search any directories but includes the specified header file.

You can include header files with either a relative or an absolute pathname. If the header filename is preceded by a colon, SC assumes it is a pathname relative to the current directory defined by the MPW `Directory` command. If the name does not begin with a colon, but contains one elsewhere, SC assumes it is an absolute pathname.

The level of nested files is limited only by the maximum number of files that MPW allows open at the same time, not including your source files and any SC temporary files. MPW generally allows a maximum of 15 nested files.

Avoiding Multiple Inclusions of Header Files

It is not uncommon to find multiple inclusions of the same header file in a single compilation unit. By default, the SC compiler does not open a header file more than once, unless you specifically direct it to do so by using the `-notOnce` option (page 3-43).

If you use the `-notOnce` option to force SC to read multiple copies of all header files, you can use `#ifndef` and `#endif` preprocessor directives around your `#include` statements to specifically exclude certain header files.

```
#ifndef __HEADERFILE__
#define __HEADERFILE__
#include myheader.h
#endif
```

If the variable `__HEADERFILE__` is already defined, the compiler will not include the file `myheader.h` because the `#include` command is skipped. This is similar to the action of the `once` pragma (page 3-49).

Data Structure Management

All data objects in C and C++ have a type and a preferred alignment. The type is defined by the ANSI language definition. The **preferred alignment** is a laying out of the elements in a data structure in memory based on the byte boundary that provides the most efficient access for the given type. The **alignment convention** defines the way in which the internal members of an application-defined aggregate type are laid out in memory. It affects the size of the aggregate type and the offset of each member from the start of the aggregate type.

The SC compiler provides a choice of three alignment conventions for data structures: PowerPC alignment, 68K alignment, and byte alignment. With **PowerPC alignment,** each element is aligned to provide the fastest memory access in the PowerPC runtime architecture. With **68K alignment,** SC matches,

bit for bit, the alignment used by the MPW C and MPW C++ compilers in the 68K runtime architecture. With byte alignment, each element is byte-aligned with no padding.

By default, SC uses 68K alignment conventions. You can change the default alignment convention with the `-align` option (page 3-7). You can also use the `options align` pragma (page 3-44) to specify different alignment conventions for individual structures, unions, and classes.

Table 2-2 describes the size and preferred alignment of data types for both PowerPC and 68K alignment. The rules for PowerPC and 68K alignment are described in *Inside Macintosh: PowerPC System Software.*

Table 2-2 Size and preferred alignment of data types

	PowerPC		68K	
Data type	Size (bytes)	Alignment	Size (bytes)	Alignment
`char`	1	1	1	1
`short`	2	2	2	2
`int`	4	4	4	4
`pointer`	4	4	4	4
`float`	4	4	4	4
`double`	8	4	8	4
`long double`	16	4	10 (12 with MC68881)	4
Enumeration	Varies	The preferred alignment of the same sized integral type	Varies	The preferred alignment of the same sized integral type

continued

Table 2-2 Size and preferred alignment of data types (continued)

	PowerPC		68K	
Data type	Size (bytes)	Alignment	Size (bytes)	Alignment
Array	Varies	The preferred alignment of the member type	Varies	The preferred alignment of the member type
Union	Varies	8, if any member is a `double` or `long double` type or the largest of the preferred alignments of the member types	Varies	2
Structure	Varies	8, if the first member is a `double` or `long double` type, or an aggregate with a preferred alignment of 8 or 4, if the largest preferred alignment of the remaining member types is 8 or the largest of the preferred alignments of the member types	Varies	2

SC Messages

SC can produce warning and error messages, both of which are sent to standard error, the file in which SC puts the generated error and warning messages. Warning and error messages appear in a format similar to this one:

```
    f(x, y);
          ^
File "badfile.c"; line 12 #Error: 1 actual arguments expected for f
#-----------------------
```

The message format lets you easily check the error or warning condition and quickly access the troublesome line of code. A typical message contains the following information:

- The erroneous line of code.
- An indicator (^) pointing to the approximate position of the error on the code line.
- The MPW file containing the erroneous line of code and the line number where it occurred. By placing your cursor at the end of this line or selecting the line and pressing the Enter key, you can automatically open the specified file as your target window with the erroneous line of code selected.
- The message type (in the preceding example, an error) and a description of the warning or error condition.
- A separator line between errors.

If a single line of source code contains multiple errors and warnings, the SC compiler repeats the line of code for each of the messages.

Warning Messages

SC issues **warning messages** to indicate usage in your source code that adheres to the language definition for the given source code dialect but that presents a potential error. As a result, SC issues a warning but does not terminate compilation. For example, the following code:

```
#include <stdio.h>

int main (void)
{
    int a, b, x = 0;

    if (a = b) { x++; }

    printf("The value is '%d'\n", x);

    return 0;
}
```

produces the following warning:

```
    if (a = b) { x++; }
              ^
File "warning.c"; line 7 #Warning 2: possible unintended assignment
#-----------------------
```

Note
You can change which warnings are issued with the `-w` option (page 3-37). ◆

Error Messages

SC issues **error messages** to indicate error conditions that you must correct in order to compile your source code successfully. If SC issues an error message, it does not produce an object file. Error conditions are typically caused by mistakes in syntax or by source code that does not adhere to the language definition.

For example, the ANSI C language definition allows no text other than a comment to follow the `#endif` preprocessor directive. If you do not follow this rule and compile your source code using strict ANSI C, you receive an error message like this one:

```
#endif NEVER
       ^
File "badfile.c"; line 2 #Preprocessor error: end of line expected
#-----------------------
```

Note

By default, SC allows four compiler errors before it terminates compilation.You can tell SC to report all error messages, no matter how many there are, by using the `-x` option (page 3-39). ◆

SC Output

In addition to producing object files, you can also use certain options to generate other types of output. You can generate reports about your source files, set how many errors and warnings SC produces, and set the location where SC puts generated object files.

- To check the syntax of your code without compiling, use the `-c` option.
- To compile code only through the preprocessor phase, use the `-e` option (which keeps comments and macro expansions in the output).
- To generate source listings of your files, use the `-l` option.
- To override the default naming conventions and directory locations of the output file, use the `-o` option.
- To control which warnings are issued, use the `-w` option. To control how many errors SC issues, use the `-x` option.
- To generate progress information about the compilation session, use the `-p` or `-v` option.
- To control the type of debugging information SC produces, use the `-sym` option.

See “Controlling Output” (page 3-28) for a complete description of these options.

Optimizations

Global optimization analyzes the code for an entire function for the possiblity of a number of optimizations: constant propagation, copy propagation, dead assignment elimination, dead code elimination, dead variable elimination, global common subexpressions, loop invariant removal, and loop induction variables. If you specify `-opt speed`, `-opt time`, or `-opt all`, the compiler will choose optimizations that save execution time over optimizations that save space. If you specify `-opt space`, the compiler will choose optimizations that save space over optimizations that save execution time.

Note

The tradeoff between speed and space is not trivial, especially in a VM environment. It may be best to use `-opt speed` only on those time critical portions of your code, and use `-opt space` for the remainder of your code. ◆

Optimization can add considerable time to a compilation, and optimized code is very difficult to debug. Use `-opt off` or `-opt none` to skip the optimization phase of the compiler.

You can control the optimization level with the `-opt` option (page 3-27).

The level of optimization you select also affects the type of inlining performed by SC. See the description of the `-inline` option (page 3-26) for more information.

CHAPTER 3

SC Reference

Contents

This chapter provides a detailed reference for the SC command line options, pragmas, and predefined symbols. It explains the general features of the compiler options and the conventions you need to follow to properly construct a compiler command line.

SC Options

SC provides a large number of command line options that allow you to control its operations. For example, you can

- specify the desired alignment of data structures
- control the level of optimizations and inlining
- generate source file listings
- specify your own include directories
- suppress unwanted warnings

Here is a typical SC command line:

```
SC -w off -ansi strict -o Hello.o Hello.c
```

This command line specifies that SC should compile the source code file `Hello.c` into the object file `Hello.o` while issuing no warnings and using strict ANSI conformance to the C language (thereby not allowing use of the Apple extensions).

In general, the available options follow these conventions:

- Options begin with a hyphen (`-`) to distinguish them from filenames.
- Options and filenames can appear in any order on the command line.
- Option names (like filenames) are not case sensitive.
- Some options require at least one parameter, which you must separate from the option by a space. For options that allow more than one parameter, you must separate each parameter from the preceding parameter with a comma and no spaces.

Note
You can repeat any option on a single compiler command line. In most cases, the last use of the option on the line sets the option state and supersedes all other uses of it on the same command line. However, you can use the `-d`, `-i`, `-xi`, `-w`, and `-sym on` (with its various modifiers) options several times on a command line, and each use sets a different value. ◆

Command line options do not override the effect of pragmas that may be in source files. For example, a `pragma options align=mac68K` pragma in a source file takes precedence over an `align power` command line option for setting the alignment of a data structure when compiling.

This section provides a detailed description of each SC command line option. The options are divided into functional groupings. For a list of the options in alphabetical order, see Table 2-1 (page 2-4).

In this section, each option description begins with a heading line that states the option name without any of its parameters. The heading line is immediately followed by a brief description of the option's function.

The option syntax is presented in several forms. Some options require that you choose one of several parameter values. In this case, the option syntax appears as shown here.

`-align` *parameter*

In a few cases, you can also assign a modifier to certain parameter values.

`-sym` *parameter*[, *modifier*]

Other options let you specify more than one parameter, with each parameter separated by a comma. In this case, the option syntax appears as shown here.

`-inline` *parameter*[, *parameter...*]

Several options include a parameter that you must replace with a specific type of information. In this case, the option syntax appears as shown here.

`-o` *pathname*

Whenever an option requires at least one parameter, a list describing the parameters follows the syntax line.

In some cases, the option has no parameter values. In this case, the syntax is simply the option name, as shown here.

```
-e
```

Language Options

You can set several compilation or language features that are not covered by the ANSI C or C++ language definitions, such as the alignment of data structures or the type of inlining SC performs.

Setting Data Type Characteristics

You can set specific characteristics of data types that you use in your application.

The `-align` option allows you to set the alignment of data structures. You can use the `-char` option to determine whether the `char` type should be signed or unsigned. You can use the `-enum` option to control the size of the `enum` type.

-align

You can use the `-align` option to determine the alignment of data structures.

`-align` *parameter*

The available values for *parameter* are

`power`	Use the PowerPC alignment rules.
`mac68k`	Use the 68K alignment rules.
`byte`	Use byte alignment with no padding.

DESCRIPTION

The `-align` option allows you to choose whether the SC compiler uses the PowerPC or 68K alignment rules for data structures as the default setting. For optimum performance in the PowerPC runtime architecture, use PowerPC alignment. For compatibility with the 68K runtime architecture, use 68K alignment.

For more information about what these alignment rules are, see “Data Structure Management” (page 2-9).

IMPORTANT

You can specify different alignment conventions for individual structures, unions, and classes using the `options align` pragma. If you prefer the finer level of alignment control provided by the `options align` pragma, do not use the `-align` option to set the default alignment convention. ▲

DEFAULT

```
-align mac68K
```

EXAMPLE

The following option specifies PowerPC alignment:

```
-align power
```

With PowerPC alignment, the size of the following data structure is 8 bytes:

```
struct S {
    short A;
    long B;
};
```

With PowerPC alignment, SC aligns `B` by placing 2 bytes of padding between `A` and `B`, because the preferred alignment for long types is 4 bytes.

The following option specifies 68K alignment:

```
-align mac68K
```

With 68K alignment, the size of the data structure is 6 bytes. Because the preferred alignment for any field bigger than a byte is 2 bytes, `B` requires no padding in order to be aligned correctly.

SEE ALSO

For more information on the differences between PowerPC and 68K alignment, see *Inside Macintosh: PowerPC System Software.*

-char

You can use the `-char` option to specify the implementation of the `char` data type.

`-char` *parameter*

The available values for *parameter* are

`signed`	Treat `char` as a signed entity.
`unsigned`	Treat `char` as an unsigned entity.
`unsignedx`	Treat `char` as a sign-extended but unsigned entity.

DESCRIPTION

The ANSI C language definition designates the representation of the built-in `char` type as implementation specific. The `-char` option determines whether objects of type `char` (including character literals) are treated as signed or unsigned quantities. In addition, you can also specify that the SC compiler treat objects of type `char` as unsigned and sign-extended.

To produce more efficient code for the PowerPC runtime architecture, use unsigned characters. To maintain compatibility with MPW C and MPW C++, use signed characters. If you use unsigned characters, the values of `CHAR_MIN` and `CHAR_MAX` in the `limits.h` header file no longer match the actual range of the `char` type.

The `unsignedx` value produces an unsigned entity, but it is sign-extended as a signed entity would be. For example, if you convert a `char` variable with a value of 0xFF into a `short` variable, and the `char` type is unsigned, the `short` type will have the value 0x00FF. If the `char` type is signed, the `short` type will have the value 0xFFFF, or –1, because the sign is extended. If the `char` type is `unsignedx`, the `short` and `char` types will both have the value 0xFFFF, but while the `short` type will still equal –1, the `char` type will equal 65535, because the variable is unsigned.

The `-char` option does not make the `char` type assignment-compatible with either the `signed char` or `unsigned char` built-in types.

Note
It is generally best to explicitly use the `signed` and `unsigned` type modifiers in your source code rather than to rely on the default value of the `char` data type. ◆

DEFAULT

```
-char signed
```

EXAMPLE

The following option specifies that the `char` data type is represented as an unsigned entity:

```
-char unsigned
```

If a variable of type `char` is unsigned, the following lines of code result in the variable `u` being assigned the value 255:

```
char c = -1;
unsigned u = c;
```

The following option specifies that the `char` data type be represented as a signed entity:

```
-char signed
```

If a variable of type `char` is signed, the same lines of code result in the variable `u` being assigned the value `MAX_UINT`.

-enum

You can use the `-enum` option to control the size of enumeration types.

`-enum` *parameter*

The available values for *parameter* are

`min`	The minimum size needed to store the values in the `enum` data type, whether one, two, or four bytes.
`int`	All `enum` types are the same size as `int` data types.

DESCRIPTION

The `-enum` option controls the size of enumeration types. The ANSI C language definition designates the size of enumeration types as implementation specific.

A minimum-sized enumeration type is the smallest size possible that represents all enumerator values of the type. Use minimum-sized enumerator types to generate the most compact data structures for your code in the PowerPC runtime architecture and to maintain compatibility with MPW C and MPW C++ (which both use minimum-sized enumeration types by default).

DEFAULT

```
-enum min
```

EXAMPLE

The following option specifies all enumeration types be of minimum size:

```
-enum min
```

Note how this value affects the following code:

```
enum small {a=0,b=255};                 /* sizeof(small) == 1 */
enum medium {c=-1, d=255};              /* sizeof(medium) == 2 because
                                            of the negative value */
enum large {e=-20,f=100000};            /* sizeof(large) == 4 */
```

Recognizing 2-Byte Asian Characters

The `-j0`, `-j1`, and `-j2` options control handling of 2-byte Asian characters.

-j0

You can use the `-j0` option to allow SC to recognize 2-byte sequences for Japanese characters.

```
-j0
```

DESCRIPTION

The `-j0` option recognizes 2-byte sequences for Japanese characters in string and character constants.

DEFAULT

As a default, the SC compiler does not recognize 2-byte Asian characters.

SEE ALSO

See the `-j1` (page 3-12) and `-j2` (page 3-13) options .

-j1

You can use the `-j1` option to allow SC to recognize 2-byte sequences for Taiwanese or Chinese characters.

```
-j1
```

DESCRIPTION

The `-j1` option recognizes 2-byte sequences for Taiwanese or Chinese characters in string and character constants.

DEFAULT

As a default, the SC compiler does not recognize 2-byte Asian characters.

SEE ALSO

See the `-j0` (page 3-12) and `-j2` (page 3-13) options.

-j2

You can use the `-j2` option to allow SC to recognize 2-byte sequences for Korean characters.

```
-j2
```

DESCRIPTION

The `-j2` option recognizes 2-byte sequences for Korean characters in string and character constants.

DEFAULT

As a default, the SC compiler does not recognize 2-byte Asian characters.

SEE ALSO

See the `-j0` (page 3-12) and `-j1` (page 3-12) options.

Setting Conformance Standards

You can set the degree to which SC conforms to two standards: the ANSI standard and the definition of the newline character used by platforms other than the Macintosh computer.

The `-ansi` option allows you to determine the degree of ANSI conformance SC requires when compiling your application. The `-noMapCR` option allows you to set the definition of the newline character to the line-feed character.

-ansi

You can use the `-ansi` option to control how closely SC should conform to the ANSI standard.

`-ansi` *parameter*

The available values for *parameter* are

`off`	Use the ANSI C or C++ language without strict typechecking.
`on`	Use the ANSI C or C++ language with strict typechecking, but allow use of Apple C language extensions.
`relaxed`	Identical to `on`.
`strict`	Like the `on` value, with the addition that the base size of `enum` data types is always the same size as an `int` type. Also, you cannot use Apple C language extensions.

DESCRIPTION

The `-ansi` option allows you to determine the degree of conformance to the ANSI C standard you want the SC compiler to require. Note that there are three levels of strictness: `off`, `on`, and `strict`, with `strict` being a superset of the `on` level.

If you use `-ansi strict`, the base size of the `enum` type is always the same as the `int` type. You can override the size of the `enum` type with the `-enum min` option. If you do this, the SC compiler warns you that there is a possible conflict.

DEFAULT

```
-ansi off
```

SEE ALSO

See the section “Language Support” (page 1-4).

-noMapCR

You can use the `-noMapCR` option to tell SC to use the line-feed character for the newline escape sequence.

```
-noMapCR
```

DESCRIPTION

The `-noMapCR` option controls the interpretation on the newline `\n` and carriage-return `\r` characters. In MPW windows or TextEdit, the newline character advances the cursor to the left margin on the next line. In TextEdit, ASCII 10 is used for the newline `\n` character code. In MPW, ASCII code 13 is used for the carriage-return `\r` code. The compiler will by default map `\n` to the value 10 and `\r` to the value 13 to conform to the MPW environment. The `-noMapCR` option will turn off the mapping of newline to carriage-return and carriage-return to newline.

Using Preprocessor Symbols

You can define preprocessor symbols. The `-d` option allows you to define one preprocessor symbol, for example, to enable debugging code.

-d

You can use the `-d` option to define a preprocessor symbol name.

`-d` *parameter*

The allowable forms for *parameter* are

name	Define the symbol *name* to have the value 1.
name=value	Define the symbol *name* to have the value *value*.

DESCRIPTION

The `-d` option defines the specified *name* to have the specified *value* or the value 1. This option is the same as using the `#define` preprocessor directive before the first line in the source file. If you use the *name*=*value* form, do not place a space either before or after the equal sign (=). Also, if *value* contains any special characters, you must enclose the *value* in double quotes (*name*="*value*"). The SC compiler removes the quotes from the value. If you want quotes to appear in the name or value, use single quotes around double quotes (for example, `'"c"'` produces `"c"`) or double quotes around single quotes (for example, `"'c'"` produces `'c'`).

The `-define` option is another name for the `-d` option.

DEFAULT

None.

EXAMPLE

The option `-d TARGET_PowerPC` is the same as the following `#define` preprocessor directive occurring before any other statement in the source file:

```
#define TARGET_PowerPC 1
```

C Language Options

There are C-specific language options that you can set to handle function prototyping and the level of typechecking that SC performs.

-proto

You can use the `-proto` option to set whether or not SC requires function prototypes.

`-proto` *parameter*

The available values for *parameter* are

`auto`	Automatically generate function prototypes based on the types of parameters.
`strict`	Require function prototypes.

DESCRIPTION

The `-proto` option allows you to set whether or not SC requires function prototypes in your source files.

If you use the `-proto auto` option, SC does not require function prototypes, but the number of parameters you use when you call a function must be the same as the number you use when you declare it.

The `-proto` option is available only when you are using the C language.

DEFAULT

```
-proto auto
```

-typecheck

You can use the `-typecheck` option to control whether SC performs strict or relaxed typechecking.

`-typecheck` *parameter*

The available values for *parameter* are

`strict`	Perform strict typechecking.
`relaxed`	Perform relaxed typechecking.

DESCRIPTION

The `-typecheck` option controls whether SC performs strict or relaxed typechecking.

The `-typecheck` option is available only when you are using the C language.

DEFAULT

```
-typecheck strict
```

C++ Language Options

There are C++-specific language options that handle template functions and classes. The `-xi` option allows you to instantiate a specific template. The `-xa` option determines the accessibility of generated template functions.

-xi

You can use the `-xi` option to instantiate a specific template.

`-xi` *template*

template The template to instantiate.

DESCRIPTION

The `-xi` option instantiates the named template.

You can use the `-xi` option several times on a command line to instantiate several templates.

DEFAULT

None.

SEE ALSO

See the `-xa` option, next.

See the `template` pragma (page 3-52).

-xa

You can use the `-xa` option to determine how accessible generated template functions are.

`-xa` *parameter*

The available values for *parameter* are

`static`	Use static scope.
`public`	Use public scope.
`extern`	Do not expand templates.

DESCRIPTION

The `-xa` option sets the scope of accessibility of the generated template functions. The functions can have static scope (that is, remain local to the file), public scope (visible outside the compiled file), or not be expanded at all.

DEFAULT

```
-xa static
```

SEE ALSO

See the `-xi` option, previous.

See the `template_access` pragma (page 3-53).

Generating Code

SC provides options that allow you to control how code is generated. You can use the `-inline` option to control inlining of functions. The `-opt` option allows you to control the types of optimizations SC performs.

-inline

You can use the `-inline` option to control the expansion of inline function within code.

`-inline` *parameter*

The available values for *parameter* are

`on`	Expand any function within a compilation unit, where inlining provides a speed optimization, including C++ functions not specified as candidates for inlining with the `inline` keyword and C functions.
`all`	Identical to `on`.
`off`	Do not perform any inline expansion of any function.
`none`	Identical to `off`.

DESCRIPTION

`-inline` is an SCpp option that controls the expansion of inline function within the code. If inlining is `all` or `on`, the compiler will attempt to replace function call with inline code if possible. When debugging C++ files, the presence of inline code can cause considerable problems when using a symbolic debugger. The specification of `-inline off` or `-inline none` will prevent inline function expansion in C++ files and a function call will be generated instead. The default is to inline code when possible.

IMPORTANT

If you specify an explicit `-inline` option on the command line, it overrides the default inlining level set by the `-opt` option. ▲

SEE ALSO

See the description of the `-opt` option, next.

-opt

You can use the `-opt` option to set the level of global optimization that SC performs.

`-opt` *parameter*

The available values for *parameter* are

`all`	Perform optimizations for `speed`, `time`, and `space`.
`speed`	Perform optimizations for highest performance.
`time`	Perform optimizations for `time`.
`off`	Perform no optimizations.
`none`	Identical to `off`.
`space`	Perform optimizations for `space`.

DESCRIPTION

The `-opt` option controls global optimization. Global optimization analyzes the code for an entire function for the possibility of a number of optimizations: Constant Propagation, Copy Propagation, Dead Assignment Elimination, Dead Code Elimination, Dead Variable Elimination, Global Common Subexpressions, Loop Invariant Removal, and Loop Induction Variables. If you specify `-opt speed`, `-opt time`, or `-opt all`, the compiler will choose optimizations that save execution time over optimizations that save space. If you specify `-opt space`, the compiler will choose optimizations that save space over optimizations that save execution time.

Note

The tradeoff between speed and space is not trivial, especially in a VM evironment. It may be best to use `-opt speed` only on those time critical portions of your code, and use `-opt space` for the remainder of your code. ◆

The optimizer can add considerable time to a compilation, and optimized code is very difficult to debug. Use `-opt off` or `-opt none` to skip the optimization phase of the compiler.

DEFAULT

`-opt off` or `-opt none` is the `-opt` default; these options skip the optimization phase of the compiler.

SEE ALSO

See the section “Optimizations” (page 2-15).

Controlling 68K Code and Data

SC controls 68K code and data utilizing MC68020, MC68030, or MC68040 instructions:

`-mc68020`

`-mc68030`

`-mc68040`

Compiling Code for the MC68020, MC68030, MC68040

You can use the SC and SCpp compilers to take advantage of MC68020, MC68030, or MC68040 instructions to produce faster and smaller code. If you use these options, the compiler generates code optimized for Macintosh computers with the MC68020, MC68030, or MC68040 microprocessor (such as the Macintosh LC, Macintosh II, or Macintosh Quadra), but this code will not run on Macintosh computers with the MC68000 microprocessor (like the Macintosh Classic, Macintosh SE, Macintosh Plus, or PowerBook 100). When you use these options, the compiler generates MC68020 instructions for bit-field operations, addressing, long word multiplication, long word division, and long word modulo.

Compiling Code for the MC68881

The following options are for the MC68881 coprocessor only.

-elems881

To optimize your code, you can use the inlines expansion of transcendental functions (such as `sin()` and `cos()`) with 68881 instructions. The `-elems881` option automatically switches on `-mc68881`.

Using the `-elems881` option also results in the automatic definition of the macro `elems68881` with a value of 1.

`elems68881`

DEFAULT

As a default, the compiler does not generate code that accesses the mc68881 coprocessor directly.

-mc68881

You can optimize the mc68881 coprocessor using the `-mc68881` option to produce faster and smaller code.

`-mc68881`

DESCRIPTION

This option changes the size of long doubles from the default of ten bytes to twelve bytes. This option also inlines floating point instructions for intrinsic functions: add, subtract, multiply, and divide.

Using the `-mc68881` option also results in the automatic definition of the macro `mc68881` with a value of 1.

DEFAULT

As a default, the compiler does not generate code that accesses the mc68881 coprocessor directly.

-paslinkage

You can use the `-paslinkage` option to request that the default linkage convention functions match the Pascal calling conventions.

```
-paslinkage
```

-b or -b2 or -b3 or -b4

You can use the `-b` or `-b2` or `-b3` or `-b4` option to produce code and data references that are relative to the position of the program counter. PC-relative object code tends to run more efficiently.

```
-b
```

or

```
-b2
```

or

```
-b3
```

or

```
-b4
```

DESCRIPTION

The PC-relative strings option is useful for code resources or other self-contained programs that contain string literals.

Table 3-1 PC-relative code options

Option	Description
`-b`	Produces PC-relative strings and function calls
`-b2`	Produces overlay strings, PC-relative strings, and function calls
`-b3`	Produces only PC-relative strings
`-b4`	Turns off `-b`, `-b2`, or `-b3`

-s segname or -seg segname

You can use the `-s segname` or `-seg segname` option to assign the object code to the specified segment name.

```
-s segname
```

or

```
-seg segname
```

DEFAULT

If you do not specify a segment name, the object code is assigned to the segment `Main`.

-model

You can use the `-model` option to specify which runtime model to use for accessing data and code.

```
-model [near|nearData|nearCode|cfmSeg|cfmFlat|far|farData|farCode]
```

DESCRIPTION

The `-model cfmSeg` and `-model cfmFlat` options request generation of the CFM-68K runtime model supporting shared libraries. These options set option `mc68020`. The `-model cfmSeg` option specifies that the code fragments can be multisegmented. The `-model cfmFlat` option specifies that the code fragments are not segmented, and code may be greater than 32K in size. The `-model cfmFlat` option also specifies that global data items of a size larger than 1200 bytes (or the specified `-auto_import size` option) are to be imported. The `-model cfmFlat` option is recommended only when building a shared library. Use the `-model cfmSeg` option when building a CFM-68K runtime application.

The `far` and `farData` options remove the 32K size limitation of the global data area, allowing it to be larger than 32K. Options `far` and `farCode` remove the 32K size restriction on code segments and the jump table, allowing them to be larger than 32K. If you use the `far`, `farData`, or `farCode` options, then you must also use the `-model far` option when linking.

The `near` and `nearData` options restrict the global data area to 32K, and options `near` and `nearCode` restrict each code segment and the jump table to 32K.

DEFAULT

```
-model near
```

SEE ALSO

See *Building and Managing Programs in MPW.*

See `-auto_import size` (page 3-27).

See `-bigseg` (page 3-26).

-onefrag

You can use the `-onefrag` option to specify to the compiler that all C++ constructs live and execute within a single fragment.

```
-onefrag
```

DESCRIPTION

Language constructs in C++ result in the SCpp compiler generating code and data modules. The compiler assumes that these modules may reside in a different fragment. The `-onefrag` option specifies to the compiler that these modules are in the same fragment and do not have to be imported or exported.

-bigseg

You can use the `-bigseg` option with `-model cfmSeg` to specify to the compiler that code is to be generated in a single code segment and may be greater than 32K. This option is automatically turned on with `-model cfmFlat`. The `-bigseg` option does not affect the `-auto_import size` option.

```
-bigseg
```

SEE ALSO

See `-model cfmFlat` (page 3-25)

-auto_import size

You use the `-auto_import size` option to increase the amount of near (32K) data space available to a CFM-68K shared library or application (refer to the *CFM-68K Runtime* manual). All data that is greater than "size" bytes will be indirectly accessed by a 32-bit pointer. Only the data pointer must lie within 32K of A5; the data itself may be sorted by the linker beyond 32K from A5.

The `-auto_import size` option instructs the compiler to treat data greater than "size" bytes as if it had been imported. Under the CFM-68K runtime model, imported data is always accessed by an indirect 32-bit pointer. Using this option is identical to surrounding data greater than "size" bytes with `#pragma import on`.

Note
This option can be used only with the `-model cfmSeg` or `-model cfmFlat` option. ◆

DEFAULT

`-auto_import 1200` is the default when `-model cfmFlat` is specified.

SEE ALSO

See `-model cfmFlat` (page 3-25).

See the `import` pragma (page 3-55)

-u

You can use the `-u` option to suppress the definition of non-ANSI predefined macros. For a list of the macros that the `-u` option affects, see Table 3-4 (page 3-59).

```
-u
```

Controlling Output

There are several options you can use to control the output you receive from the SC compiler. You use these options to get information about your source files during compiling, to set warning and error levels used by SC, and to create export lists of the names of functions and objects within external linkage.

Controlling Compilation Output

You use the options in this section to control the compilation output as follows:

- The `-c` option allows you to check the syntax of your code without generating an object file.
- The `-e` option allows you to stop the compilation of your files after the preprocessor phase.
- The `-l` option provides you with a listing of your source files with the include files expanded.
- The `-o` option allows you to choose the name for the object file created by the compilation.
- The `-p` option allows you to obtain a progress report on the compilation of your source files.
- The `-frames` option allows the computer to generate stack frames.
- The `-sym` option lets you control whether or not SC generates debugging information for your source file.
- The `-nomfmem` option allows you to specify that MultiFinder memory not be used for processing.
- The `-mbg` option generates MacsBug information.
- The `-trace` option lets you control the generation of function preambles and postambles for the MPW debuggers or profiler.
- The `-v` option allows you to generate progress information about the compilation.

-c

You can use the `-c` option to check the syntax of your code without generating an object file.

```
-c
```

DESCRIPTION

The `-c` option allows you to make sure that your code is syntactically correct without generating an object file.

DEFAULT

The default is to generate an object file.

SEE ALSO

If you want an object file, the `-o` option (page 3-31) controls the name of the resulting file.

-e

You can use the `-e` option to perform macro expansion and include specified header files in the output.

```
-e
```

DESCRIPTION

The `-e` option performs macro expansion and includes the specified header files in the resulting output. When you use the `-e` option, comments remain in the output and the SC compiler does not perform semantic analysis and code generation, which occur after the preprocessing phase.

You must use the `-c` option (page 3-29) to stop object generation.

IMPORTANT

With SC, the `-e` option does not automatically send the results of the preprocessor phase to standard output as it does with some compilers. You must use the `-e` option with the `-l` option to see the results. ▲

DEFAULT

None.

SEE ALSO

See the description of the `-l` option, next.

-l

You can use the `-l` option to generate a listing of your source file with the include files expanded.

`-l` *filename*

filename	The name of the listing file. You must specify a filename; there is no default.

DESCRIPTION

The `-l` option generates a source file listing and sends it to the file you specify. If the listing includes preprocessor output (as generated by the `-e` option), it shows macro expansion and included header files. The name of the included header file appears after the included text.

If the listing does not include preprocessor output, it does not contain expanded macros or included header files.

You must specify the name of the listing file that you want SC to generate. A typical filename suffix is `.lst`.

Note

The generated listing does not include line numbers. ◆

DEFAULT

None.

SEE ALSO

See the description of the `-e` option, previous.

-m

Used to generate 32-bit references for data (currently not implemented).

-o

You can use the `-o` option to override the default naming conventions and directory location that SC uses for generated object files.

`-o` *filename*

filename Any filename or directory name.

DESCRIPTION

The `-o` option specifies the output filename for the generated code. If you specify only a directory and not a filename, SC still assigns a default filename to the object file, but puts it in the specified directory. Any directory that you specify in a pathname must already exist. SC will not create new directories for you.

If the `-o` option is not specified, the output file has the same name as the input file with the extension `.o` added (for example, `hello.c.o`). If a directory is specified, the output file has the same name as the input file with the extension `.o` added, except SC places the object file in the specified directory.

DEFAULT

For a filename, SC appends a `.o` extension to the source filename (for example, `myfile.c` becomes `myfile.c.o`) and places the file in the same directory as the source file.

EXAMPLE

The following command names the object files resulting from the compilation:

```
SC -o myfile.o newname.c
```

The command

```
SC -o ::objects: myfile.c
```

sends the resulting object file to the following directory location:

```
::objects:myfile.c.o            /* for myfile.c */
```

-p

You can use the `-p` option to generate progress information about the compilation.

```
-p
```

DESCRIPTION

The `-p` option emits progress information to standard output. Progress information tells you which files SC is including, which functions it's processing, and the number of instructions and code bytes generated.

DEFAULT

No progress information is generated.

EXAMPLE

The following command produces progress information about the file `example.c`:

```
SC -p example.c
```

You receive one progress report after the compilation, as shown here:

```
sc -p example.c
C Compiler 8.0.3 Copyright (C) 1985-1995 by Symantec Corporation
             (written by Walter Bright, adapted to the 68000 by Symantec)
'example.c'
        'Stonewall:MPW:Interfaces:CIncludes:stdio.h'
        'Stonewall:MPW:Interfaces:CIncludes:NullDef.h'
        'Stonewall:MPW:Interfaces:CIncludes:SizeTDef.h
        'Stonewall:MPW:Interfaces:CIncludes:SeekDefs.h:
        'Stonewall:MPW:Interfaces:CIncludes:VaListTDef.h'
    gus
    main
```

-frames

If the `-frames` option is on, the compiler will always generate stack frames, even for functions that normally wouldn't have one. If a function has no local variables and takes no parameters, the compiler does not generate a stack frame for it, unless this option is on.

The compiler does not usually create stack frames for C functions that have no arguments or local variables. If you don't use this option, `-mbg on` doesn't give names to functions; it just lists their hex addresses. Debugging with a low-level debugger like MacsBug is much more difficult without function names.

```
-frames
```

-sym

You can use the `-sym` option to control whether SC generates debugging information for your source file.

`-sym` *parameter[,modifier]*

The available values for *parameter* are

`off`	Suppress generation of debugging information.
`on`	Generate debugging information only for referenced types.
`full`	Identical to `on`.

The *modifier* is for use only with the `on` (or `full`) parameter value and can have one or more of the following values:

`nolines`	Suppress the generation of source line information.
`notypes`	Suppress the generation of information about the data types of variables.
`novars`	Suppress the generation of information about variables.
`alltypes`	Generate debugging information for all types, even if they are not represented in the compilation.

DESCRIPTION

The `-sym` option specifies the desired level of debugging information created by SC.

Using the `alltypes` modifier significantly increases the size of the output.

IMPORTANT

The `-sym` option with either the `on` or `full` parameter value automatically suppress all SC optimizations. ▲

DEFAULT

```
-sym off
```

-nomfmem

By default, the compiler uses MultiFinder memory if it is available. You can use the option `-nomfmem` to specify that the compiler should not use MultiFinder memory for processing.

```
-nomfmem
```

-mbg

Output symbols for `-mbg` (MacsBug). This option takes the following arguments. The default is `full`.

```
-mbg [on|full|off]
```

DESCRIPTION

When the `-mbg` option is used without an argument, the compiler assumes `-mbg on`.

The `-mbg on` option generates MacsBug names if the compiler has created a stack frame.

The `-mbg full` option generates MacsBug names if the compiler has created a stack frame.

The `-mbg off` option prevents generation of output for MacsBug.

DEFAULT

`-on` and `-full` are the default (both are the same).

-trace

You can control the generation of function preambles and postambles for the MPW debuggers or profiler using the `-trace` option. The `-trace` option supports generating performance metrics using the MPW Performance Library. These preambles and postambles let you use the MPW Performance-Measurement Tools by inserting calls to the Performance Library before and after each function.

```
-trace [on|off|always|never]
```

For more information on the MPW Performance-Measurement Tools, see *Building and Managing Programs in MPW*.

DESCRIPTION

The `-trace on` option generates an output preamble and postamble unless a `#pragma trace off` is encountered.

The `-trace off` option does not generate an output preamble or postamble unless a `#pragma trace on` is encountered (default).

The `-trace always` option always generates an output preamble and postamble.

The `-trace never` option never generates an output preamble or postamble.

-v

You can use the `-v` option to generate progress information about the compilation.

```
-v
```

DESCRIPTION

The `-v` option emits progress information to standard output. Progress information tells you which files SC is including, which functions it's processing, and the number of instructions and code bytes generated.

DEFAULT

No progress information is generated.

EXAMPLE

The following command produces progress information about the file `example.c`:

```
SC -v example.c
```

You receive one progress report after the compilation, as shown here:

```
sc -v example.c
C Compiler 8.0.3 Copyright (C) 1985-1995 by Symantec Corporation
             (written by Walter Bright, adapted to the 68000 by Symantec)
'example.c'
        'Stonewall:MPW:Interfaces:CIncludes:stdio.h'
        'Stonewall:MPW:Interfaces:CIncludes:NullDef.h'
        'Stonewall:MPW:Interfaces:CIncludes:SizeTDef.h
        'Stonewall:MPW:Interfaces:CIncludes:SeekDefs.h:
        'Stonewall:MPW:Interfaces:CIncludes:VaListTDef.h'
    gus
main
```

Setting Warning and Error Levels

You can control the types of warnings you receive and the number of errors SC gives you. The `-w` option allows you to control which warnings SC issues and whether the compiler issues them as warnings or as errors. The `-x` option allows more than the default of four errors without terminating the compilation.

-w

You can use the `-w` option to control which SC warning messages are issued.

`-w` *parameter*

The available values for *parameter* are

[`is`]`err`[`or`]	Treat all warnings as errors.
`off`	Suppress all warning messages.
value[,...]	Suppress only the specified warnings.

DESCRIPTION

The `-w` option controls which SC warning messages are issued. You can choose to treat warnings as errors, suppress all warning messages, or suppress specific warnings.

Each warning has a value assigned to it. If you want to suppress only certain warnings, you can specify their values as parameters of the `-w` option.

The values for the warnings issued by the SC compiler for C and C++ language compiles are as follows:

Value	Description
1	Function is too complicated to inline.
2	There is possible unintended alignment.
3	You cannot nest comments.
5	There is no tag name for a `struct` or `enum` type variable.
6	The value of the expression is not used.
7	There is a possible extraneous semicolon.
8	There is a very large automatic variable.
9	The expression was ignored because you used `delete[]` instead of `delete[`*expression*`]`.
10	The `operator++()` or `operator--()` should have been `operator++(int)` or `operator--(int)`.

continued

Value	Description
11	A nonconstant was initialized to a temporary variable.
12	A variable is used before it is set.
13	A floating-point constant is out of range.
14	Function definitions with separate parameter lists are obsolete in C++.
15	The returning address of an automatic variable will be gone as soon as the function returns.
17	The pragma is unrecognized.
18	You are casting from an incomplete structure type.
19	The shift value is too large or is negative.
21	The `-sym` option suppresses all optimizations.
22	The `-sym` option suppresses all inlining of functions.

DEFAULT

The default is for SC to issue all warnings as warnings.

EXAMPLE

You can suppress the unintended alignment and extraneous semicolon warnings using the following:

```
SC -w 2,7 example.c
```

-x

You can use the `-x` option to allow more than the default maximum number of errors (four) without terminating the compilation.

```
-x
```

DESCRIPTION

The `-x` option causes compilation to continue no matter how many errors are found.

DEFAULT

The default is for SC to generate only four errors before terminating compilation.

Setting Precompiled Header Options

You can use the `-dump` option to save the state of compilation in a file and the `-load` option to load that state of compilation back into SC. You can use these options to dump and load definitions read from header files quickly. This can shorten compilation times.

-dump

You can use the `-dump` option to save the state of the compilation in a file that you specify.

`-dump` *filename*

filename	The name of the file in which you want to save the state of compilation.

DESCRIPTION

The `-dump` option saves the state of compilation in a file you specify. Only declarations and variable definitions are allowed in a dump file; no code is included.

You can use the `-dump` option to create an artificial file, which includes a set of headers that you want to include elsewhere, in order to shorten compilation times.

A dump file created by the SC compiler is *not* compatible with the SCpp compiler and vice versa.

Note
Another form of the `-dump` option is the `-dumpc` option. These options are equivalent. ◆

DEFAULT

The state of the compilation is not saved.

SEE ALSO

See the description of the `-load` option, next.

-load

You can use the `-load` option to restore a saved state of compilation from a file you specify.

`-load` *filename*

filename	The name of the file in which you have saved the state of compilation.

DESCRIPTION

The `-load` option loads a file in which you have saved the state of compilation into the SC compiler.

A dump file created by the SC compiler cannot be loaded into the SCpp compiler and vice versa.

Note
Another form of the `-load` option is the `-loadc` option. These options are equivalent. ◆

DEFAULT

None.

SEE ALSO

See the description of the `-dump` option, previous.

Setting Header File Options

You can use the `-i` option to set the list of directories SC searches to locate header files. You can use the `-notOnce` option to allow SC to read in multiple copies of the same header file.

You can use the `-fatext` option to specify that when the output file name is generated by the compiler, the extension (`.68k.o`) appended to the input file name should indicate the compiler target.

-fatext

You can use the `-fatext` option to specify that the compiler should use target-unique extensions for output filenames.

DESCRIPTION

The `-fatext` option specifies that when the output filename is generated by the compiler, the extension (`.68k.o`) appended to the input filename should indicate the compiler target.

The `-fatext` option is primarily available to allow 68K and SCpp files to exist in the same directory and be differentiated by filename extension.

-i

You can use the `-i` option to add a directory to the list of directories that SC searches to locate header files included in your source file.

`-i` *pathname[,pathname]*

pathname The directory to search.

DESCRIPTION

The `-i` option allows you to add the specified directory to the list of directories in which the SC compiler looks for included source files. You can specify more than one `-i` option on the same command line. SC searches for directories named in the `-i` option in the order in which they appear on the command line.

DEFAULT

None.

EXAMPLE

The following command tells SC to search for any included header files in the directory `::my_includes` before searching the directory `{CIncludes}`.

```
SC -i ::my_includes myfile.cp
```

SEE ALSO

See the section “Including Header Files” (page 2-8).

-notOnce

You can use the `-notOnce` option to allow SC to read multiple copies of header files.

```
-notOnce
```

DESCRIPTION

The `-notOnce` option instructs the SC compiler to read multiple copies of all header files. You cannot use the `-notOnce` option to specify a certain header file.

DEFAULT

The default is for SC to read header files only once.

SEE ALSO

See the `once` pragma (page 3-49), which allows you to specify directly in a header file that SC should include it only once.

Pragmas

Pragmas are an ANSI C language standard feature that let you include in your source files directives to a specific compiler while still producing ANSI-compliant code. You generally use pragmas to control compilation aspects that are outside the scope of the ANSI standard—for example, the target processor, optimizations, and debugging information. A compiler ignores any pragmas that it does not recognize.

options align

You can use the `options align` pragma to specify PowerPC or 68K alignment for data structures.

`#pragma options align=`*parameter*

The available values of *parameter* are

`power`	Use PowerPC alignment.
`mac68k`	Use 68K alignment.
`reset`	Return to the previous alignment.

DESCRIPTION

The `options align` pragma lets you specify the desired alignment of data structures in your program. Unlike the `-align` option, which sets the default alignment convention for all data structures in a program, you can use the `options align` pragma to specify different alignment conventions for individual structures, unions, and classes. The pragma affects all the following code until another `options align` pragma is encountered.

In general, you'll use PowerPC alignment for optimum performance in the PowerPC runtime architecture. You might want to use 68K alignment, however, if you want data structures to be bit-for-bit compatible with 68K data structures created by 68K applications or by emulated 68K code on the PowerPC processor.

When you place the `options align` pragma in your source code, do not add any white space before or after the equal sign. Also, make sure that the source code that follows the pragma begins on a new line (not on the same line as the pragma).

IMPORTANT

The parameter names for the `options align` pragma are case sensitive. ▲

You should apply the `options align` pragma only to complete declarations of structures, unions, and classes, which are declarations that define all the fields of the structure. If you apply the pragma to incomplete declarations (which define the structure by name but do not immediately define its fields) the `options align` pragma has no affect. SC will instead align the structure using the alignment in effect when the field definitions appear. Here is an example of an incomplete declaration:

```
#pragma options align=mac68k
struct later_gator;
#pragma options align=reset

#pragma options align=power
struct later_gator {
    double magic_double;
};
#pragma options align=reset
```

In this case, SC ignores the alignment conventions in effect where the incomplete declaration of the `later_gator` structure appears and instead uses the alignment mode in effect where the complete declaration of `later_gator` appears.

Place the `options align` pragma directly before the structure, union, or class declaration for which you want to specify alignment. Do not place the `options align` pragma anywhere between the braces of a structure, union, or class

declaration. Doing so will have undefined effects. For instance, consider this illegal usage:

```
#pragma options align=power
struct Stuff {
    short s;
#pragma options align=mac68k
    struct ForToolbox {
        unsigned char aa;
    } ft;
#pragma options align=reset
    int i;
};
#pragma options align=reset
```

To create a legal version of the same code, declare the inner structure before the outer structure, as shown here:

```
#pragma options align=mac68k
    struct ForToolbox {
        unsigned char aa;
    };
#pragma options align=reset

#pragma options align=power
struct Stuff {
    short s;
    struct ForToolbox ft;
    int i;
};
#pragma options align=reset
```

You must consistently define the same alignment for a given structure, class, or union declaration throughout your application. If a structure, class, or union appears in several places in your application, do not define it to have 68K alignment in one occurrence and PowerPC alignment in another occurrence. Define only one alignment convention for a single statement of source code.

Note
The alignment of an array is always determined by its element type. Scalar types use 68K alignment, regardless of the type of alignment convention in effect, unless they appear inside aggregate types. In this case, SC aligns the scalar types according to the alignment convention in effect. ◆

DEFAULT

The default is 68K alignment.

EXAMPLE

In the following example, the `options align` pragma is applied to several structures:

```
#pragma options align=mac68k
const struct mac68k3chars {
    char a, b, c;
};
#pragma options align=reset

#pragma options align=power
const struct power3chars {
    char a, b, c;
};
#pragma options align=reset
```

The `options align` pragma first specifies 68K alignment conventions for the `mac68k3chars` structure. As a result, the `a`, `b`, and `c` fields are at offsets 0, 1, and 2, respectively, and there is a 1-byte pad at offset 3. The size of the structure is 4, and its preferred alignment is 2. The `options align` pragma then specifies PowerPC alignment conventions for the `power3chars` structure. As a result, its `a`, `b`, and `c` fields are at offsets 0, 1, and 2 but have no padding at the end. The size of the structure is 3, and its preferred alignment is 1.

This next example shows a PowerPC structure nested in a 68K structure:

```
#pragma options align=power
struct powerLittle {
    char ch;
    int fi;
};
#pragma options align=reset

#pragma options align=mac68k
struct mac68kBig {
    char f1;
    struct powerLittle f2;
    char f3[3];
    int f4;
};
#pragma options align=reset
```

The `options align` pragma first specifies PowerPC alignment conventions for the `powerLittle` structure. As a result, the `ch` field starts at offset 0, there is a 3-byte pad at offsets 1 through 3, and the `fi` field starts at offset 4. The size of the structure is 8, and its preferred alignment is 4. Next, the `options align` pragma specifies 68K alignment conventions for the `mac68kBig` structure. As a result, the `f1` field starts at offset 0, there is a 1-byte pad at offset 1, and the `f2` field starts at offset 2.

Note
The `powerLittle` structure does not receive its preferred alignment of 4 because all structures that appear inside a `mac68k` structure are aligned to 2 bytes, regardless of their preferred alignment. The `f3` field begins at offset 10 and continues for 3 bytes. There is a 1-byte pad at offset 13. The `f4` field starts at offset 14. The size of the structure is 18, and its preferred alignment is 2. ◆

SEE ALSO

See the section “Data Structure Management” (page 2-9), and the description of the `-align` option (page 3-7).

noreturn

You can use the `noreturn` pragma to tell SC that the specified function does not return.

`#pragma noreturn` *function-name*

function-name The function.

DESCRIPTION

The `noreturn` pragma specifies that the function does not return. The pragma must be declared at the end of the function's declaration. The `noreturn` pragma can improve optimization of your application.

DEFAULT

None.

once

You can use the `once` pragma to include the file only once.

`#pragma once`

DESCRIPTION

The `once` pragma instructs SC to include the file only once in the compilation, even if the file is included in several source files. This pragma is largely for compatibility with other compilers because SC always includes any file included in a precompiled header only once. You can explicitly instruct SC to include a file more than once using the `-notOnce` option (page 3-43).

DEFAULT

SC includes a file only once.

message

You can use the `message` pragma to print the specified text as a warning.

```
#pragma message "text"
```

text The text of the warning message.

DESCRIPTION

The `message` pragma prints the specified text at a specific moment during compilation. For example, if your code contains a complex `if` statement, you can insert a message pragma to print a warning if a particular condition tested by preprocessor conditionals is `true`.

options

You can use the `options` pragma to use command-line options as pragmas.

```
#pragma options(parameter,...)
```

DESCRIPTION

The `options` pragma lets you set the value of certain options to either `true` or `false`. You can set any number of these options when you use this pragma. In order to set the option to `true`, you use one of the parameter values from either Table 3-2 or Table 3-3.

You can test the settings of these options using `__option(`*parameter*`)`. For example, you can test whether the `ansi` option is on or not, using the following:

```
if __option(ansi) {
    ...
}
```

Table 3-2 shows all parameter values for the `options` pragma, except for those used with warnings, which are shown in Table 3-3.

Table 3-2 Parameter values for the `options` pragma

Parameter	Related option
`ansi`	The `-ansi` option (page 3-14)
`ansi_relaxed`	The `-ansi` option (page 3-14)
`ansi_strict`	The `-ansi` option (page 3-14)
`read_header_once`	The `-notOnce` option (page 3-43)
`chars_unsigned`	The `-char` option (page 3-9)
`pack_enums`	The `-enum` option (page 3-11) with the `min` parameter value
`struct_align`	The `-align` option (page 3-7)
`stop_at_first_err`	The `-x` option (page 3-39)
`report_all_err`	The `-x` option (page 3-39)
`generate_warn`	The `-w` option (page 3-38)
`stdc`	The `-ansi` option (page 3-14)
`trigraphs`	The `-ansi` option (page 3-14)
`thinkc`	The `-ansi` option (page 3-14)
`objectc`	The `-ansi` option (page 3-14)
`signed_pstrs`	The `-ansi` option (page 3-14)
`require_protos`	The `-proto` option (page 3-16)
`infer_protos`	The `-proto` option (page 3-16)
`mapcr`	The `-noMapCR` option (page 3-15)

Table 3-3 gives the parameter values for warnings you can use with the `options` pragma. Warnings are explained in the description of the `-w` option (page 3-38).

Table 3-3 Parameter values for warnings

Parameter
`warn_unintended_assign`
`warn_nest_comments`
`warn_struct_without_tag`
`warn_unused_expressions`
`warn_large_auto`
`warn_used_before_set`
`warn_return_addr_auto`
`warn_unrecognized_pragma`

template

You can use the `template` pragma to create one or more instantiations of a template.

`#pragma template` *class<arg1, arg2, ...>*

`#pragma template` *function(arg1, arg2, ...)*

DESCRIPTION

The `template` pragma creates one or more instantiations of a template.

The `template` pragma is for the C++ language only.

EXAMPLE

Consider the following template definition:

```
template <class T>T TheTemplate(T *t1)
{
    if (t1) return *t1;
    return 0;
}
void example (int x)
{
    TheTemplate(&x);
}
```

You could expand this template by using either the `template` pragma or by using the command line option `-xi` (page 3-18).

```
#pragma template TheTemplate(int *)

-xi 'TheTemplate(int *)'
```

SEE ALSO

See the `template_access` pragma, next.

See the `-xi` option (page 3-18).

template_access

You can use the `template_access` pragma to control the type of access to template expansions.

`#pragma template_access` ***parameter***

The available values of *parameter* are

`public`	Expand the template and export it to other files. You must specify which template, using the `template` pragma (page 3-52).
`extern`	Do not expand the specified templates during compilation.
`static`	When expanding templates, keep them local to the current source file.

DESCRIPTION

The `template_access` pragma controls the type of access to template expansions.

You can use the `public` value to expand a template and export it to other files. You must specify the template with the `template` pragma.

You can use the `extern` value to export but not expand the template.

You can use the `static` value to expand the template but not export the expansion from the file.

The `template_access` pragma is for the C++ language only.

DEFAULT

The default is static access.

EXAMPLE

You can set the access of the template to public access as follows:

```
#pragma template_access public
```

SEE ALSO

See the `template` pragma (page 3-52).

See the `-xa` option (page 3-19).

New Pragmas

The SC and SCpp compilers support three pragmas that specify routines or variables to be imported or exported from other code fragments (such as import libraries) during the link procedure. These pragmas are

- `import`, which specifies symbols to be imported from another fragment
- `export`, which specifies symbols that will be exported from the current fragment
- `internal`, which specifies symbols that will be referenced only in the current fragment

In many cases, commonly used symbols are already tagged with the appropriate pragmas in Apple Computer's universal header files. If you are using standard shared libraries, you should check the header files before adding pragmas to your code.

Note
The `import` and `export` pragmas supersede the older `lib_export` pragma. Although the SC and SCpp compilers currently support `lib_export`, later versions may not, so you should use `import` and `export`. ◆

import

You can use the `import` pragma to tell the compiler which variables or routines must be imported from other code fragments, thus eliminating possible unresolved reference errors.

`#pragma import on|off`

`#pragma import list` *name1 [,name2]...*

DESCRIPTION

In the first syntax form (`#pragma import on|off`), the compiler tags all symbols defined or declared after `#pragma import on` as imports until it encounters the `#pragma import off` statement. In the second form (`#pragma import list` *name1 [,name2]...)*, you list by name the code and data items you want to have imported.

The CFM-68K runtime compilers assume that all functions reside in the current fragment, so you must use the `import` pragma to mark any that are cross-fragment (that is, those functions whose definition resides in another fragment, such as a shared library). The compiler then creates indirect references to all functions declared as imported.

The CFM-68K runtime compilers also assume that all data items reside in the current fragment. This is different from the PowerPC compilers, which assume that all data items are cross-fragment. Data items defined outside the current fragment must be declared as such using the `import` pragma. The pragma creates an indirect reference to the data, which can be resolved by the linker.

IMPORTANT

References to imported functions and data items must be indirect. The linker cannot convert a direct reference to an indirect one, so a direct reference to an imported symbol will result in a linker error. Therefore, you must use the `import` pragma to define all imported functions and data items. ▲

You should generally use the `import` pragma on symbol declarations rather than definitions. You should put the `pragma import` statement in a public header file so that any routine that needs to can access the imported symbols. While developing the code fragment, you can turn off `pragma import` with an `#ifdef` conditional statement to improve local code generation.

export

You can use the `export` pragma to tell the compiler which routines or data items will be exported from the fragment being created. For example, shared libraries declare many of their symbols as exported, since (by definition) they will be called by routines from other fragments.

```
#pragma export on|off
```

```
#pragma export list name1 [,name2]...
```

DESCRIPTION

As with the `import` pragma, you can either declare or define the exported symbols between the `#pragma export on` and `#pragma export off` statements or list them with `#pragma export list`.

Aside from marking symbols to be exported, the `export` pragma has no effect on the generated code or data. The `export` pragma has no effect on static functions or data items.

Unlike the `import` pragma, you should generally use the `export` pragma on definitions, rather than declarations. You should place this pragma in source code files, rather than public header files, unless you conditionalize the header file to activate the pragma only during development of the fragment that defines the symbol.

internal

You can use the `internal` pragma to limit the scope of the defined functions or data items to the fragment that contains them. Items declared `internal` cannot be exported to, or imported from, other fragments. The syntax for the `internal` pragma is identical to those for the `import` and `export` pragmas.

```
#pragma internal on|off
```

```
#pragma internal list name1 [,name2]...
```

DESCRIPTION

The `internal` pragma allows optimizations that might otherwise violate language semantics. For example, routines declared `internal` do not have an XVector and can use local calling conventions. For further information on XVectors, refer to *Building CFM-68K Runtime Programs for Macintosh Computers.*

Note
The `internal` pragma has no effect on static variables since they are implicitly internal. However, the `internal` pragma allows the compiler to omit the XVector for a file-scoped routine (since it will never be called via a pointer). You cannot take the address of an `internal` routine, because it does not have an XVector. ◆

You must declare internal routines and data items no later than the definition and before you reference them.

The `internal` pragma is well suited for situations where you cannot apply the standard `static` declaration. For example, if you have variables and routines that are referenced from multiple source files (and therefore cannot be declared `static`), you can still ensure optimum code generation by declaring them `internal` in a private header.

You should not use the `internal` pragma in public headers, except when applied to variables during an internal build. In that case, you should make the pragma declarations conditional using `#ifdef` statements.

Pragma Examples

The following header listings show how you might use pragmas when writing your code.

Listing 3-1 A public header `Library.h`

```
#include <IncludeFile.h>

#pragma import on
extern int mooInt;
extern void libProc(void);
#pragma import off
```

The public header file indicates symbols imported from another code fragment.

Listing 3-2 A private header `Internal.h`

```
#include <Library.h>

#pragma export list mooInt, libProc

#pragma internal on
int PrivInt;
void PrivProc(void);
#pragma internal off
```

The private header file exports the symbols `mooInt` and `libProc` for public use, but restricts access of `PrivInt` and `PrivProc` to source files that include `Internal.h`.

IMPORTANT

If you are writing C++ code, you should use the `on|off` form of the pragmas to avoid name-mangling issues. The `list` form does not support mangled names or full-function method signatures. ▲

Pragma Compatibility Issues

The `import` and `export` pragmas are completely independent of each other. You can declare symbols as both imported and exported without running into compile or link problems.

The `import` and `internal` pragmas, however, are mutually exclusive. If you declare a symbol as imported, it cannot also be declared internal and vice versa. Activating one pragma implicitly deactivates the other, but deactivating one does not implicitly activate the other. For example, if the function `mooCow` is imported, it obviously cannot be internal. However, if you later remove the `import` tag, this does not mean that `mooCow` is now internal.

The `export` and `internal` pragmas are independent when used to declare variables, but not functions. Exported routines need an XVector to export, but the `internal` declaration removes this requirement.

Predefined Symbols

The SC compiler supports the predefined symbols specified by the ANSI C language definition, as well as its own predefined symbols for use in conditional compilation. Table 3-4 lists the ANSI predefined symbols.

Note
The predefined symbols use a double underline "__" at the beginning of the symbol, a single underline "_" in the middle, and another double underline "__" at the end of the symbol, as shown in Table 3-4 and Table 3-5. ◆

Table 3-4 ANSI predefined symbols

Symbol	Predefined use
`__DATE__`	The date of compilation of the source file.
`__TIME__`	The time of compilation of the source file.
`__LINE__`	The line number of the current source line (a decimal constant).

continued

Table 3-4 ANSI predefined symbols (continued)

Symbol	Predefined use
`__FILE__`	The current source or `include` file filename.
`__STDC__`	Value is 0 if `-ansi off` is set.
	Value is 1 if compiling in strict ANSI C (`-ansi strict` or `-ansi on`).
`__FPCE__`, `__FPCE_IEEE__`	Value is 1 to indicate conformance with the NCEG or IEEE standards.

Table 3-5 shows the predefined symbols specific to the SC compiler.

Table 3-5 SC predefined symbols

Symbol	Predefined meaning
`macintosh`	Code is compiled for Macintosh
`MC68000`	Code is compiled for Macintosh.
`mc68000`	Code is compiled for Macintosh.
`m68K`	Code is compiled for Macintosh.
`__SC__`	The hex version number of SC.
`THINK_PLUS`	The hex version number of SC.
`__cplusplus`	Value is 1 if compiling in the SCpp compiler.
	The value is not defined if compiling in the SC compiler.
`mc68881`	Value is 1 if the option for generating 68881 instructions is set.

continued

Table 3-5 SC predefined symbols (continued)

Symbol	Predefined meaning
`__FAR_CODE__`	Value is 1 if the `-model` option is either `far` or `farCode`.
`__FAR_DATA__`	Value is 1 if the `-model` option is either `far` or `farData`.
`__CFM68K__`	Value is 1 when you specify the `-model cfmSeg` or `-model cfmFlat` option during compilation.

APPENDIX

ANSI C and C++ Language Restrictions

This appendix describes the C and C++ language restrictions that you should be aware of if you decide to use the `-ansi on` or `-ansi strict` options. The `-ansi strict` option is a superset of the `-ansi on` option, with two additional restrictions: Apple C language extensions cannot be used, and the base size of `enum` data types is always the same size as an `int` type.

ANSI Restrictions on C and C++ Languages

There are basic ANSI restrictions that apply to both the C and C++ languages.

- The base size of the `enum` type is determined by the minimum size needed to store the values in the type.
- The following keywords are not recognized: `__handle`, `_inf`, `__nan`, `__nans`, `__machdl`, `asm`, and `pascal`.
- Predefined macros that do not start with a single or double underscore are not defined.
- You cannot use arithmetic on pointers to functions.
- Trigraphs are supported. **Trigraphs** are sequences of three letters that are treated as one. The sequence is `??` and an additional character. Trigraphs let computers without characters such as braces (`{` and `}`), tildes (~), and carets (^) use C++. However, many Macintosh applications use character literals that resemble trigraphs. For example, the file type `'????'` is interpreted as `'?^'`. To write the file type `'????'`, use `'???\?'`.
- Text on the end of a preprocessor line is not ignored and is an error. For example, `#endif COMMENT` needs to be `#endif /* COMMENT */`.
- You cannot use binary numbers such as 0b10110.
- At least one hexadecimal number must follow a `\x` escape sequence.

- The program must end with a new line.
- You cannot get the size of a function using the `sizeof` operator.
- You cannot use the unordered, floating comparisons operators `!`, `<>=`, `<>`, `<>=`, `!<=`, `!<`, `!>=`, `!>`, and `!<>`.
- You cannot use hexadecimal floating-point constants.
- A non-integer expression is not converted to an integer expression where a constant expression is required.
- The type of a `case` expression cannot be larger than the type of the `switch` expression.
- Function prototypes for arguments must match the function definitions in number and type.
- You cannot put a `sizeof` operator or a `cast` operator in a preprocessor expression.
- You cannot use dimensionless arrays as the last member of `struct` definitions.
- Empty member lists in `enum` declarations and member lists with a trailing comma are syntax errors.
- You cannot place a `void` expression in a logical `&&` or `||` expression.
- Any `case` label constants must be of type `int` or `unsigned int`.
- Declarators must declare at least one variable.
- Function prototypes with ellipsis (…) must have at least one other argument.
- Macros must match previous definitions exactly when being redefined.
- You cannot put commas in constant expressions.
- Unrecognized # directives are in error.

ANSI C-Specific Restrictions

There are ANSI C language-specific restrictions.

- You cannot use C++-style comments in C code. C++-style comments use the double slash (`//`) format.
- You cannot have an empty `struct` or `union` definition.
- A function declared with ellipsis (...) does not match a function declared without ellipsis.
- Old-style function definitions must match promoted size of the arguments.

ANSI C++-Specific Restrictions

There are also ANSI C++-specific restrictions.

- Anonymous unions must be static.
- Member functions cannot be static.
- You cannot generate a reference to a temporary variable.
- You cannot convert to and from a `void` type.
- You cannot use the preincrement operator function or the postincrement operator function as an overloaded function for postincrement or postdecrement.
- Template class instantiations cannot introduce new nonlocal names.
- You cannot use `#ident`.
- You cannot cast an `lvalue` type to a different type.
- You cannot type something `void` where a value is required.
- The type `void *` is not compatible with other pointer types.

- Function declarations with separate parameter lists never match functions with supplied prototypes.
- String literals used to initialize arrays of type `char` are always null-terminated.
- You cannot have function definitions with separate parameter lists.

Glossary

64-bit format A format for the `long double` type that stores floating point values of up to 15- or 16-decimal digit precision and uses the same format as the `double` type. See also **128-bit format.**

128-bit format A format for the `long double` type that consists of two double-format numbers. It has the same range as 64-bit (`double`) format but much greater precision. Also known as the `double double` type. See also **64-bit format.**

68K Any member of the Motorola 68000 family of microprocessors.

68K alignment An alignment convention supported by SC that matches, bit for bit, the alignment used by the MPW C and MPW C++ compilers in the 68K environment. See also **PowerPC alignment** and **alignment convention.**

68K application An application that contains code only for a 68K microprocessor. See also **fat application.**

68K-based Macintosh computer Any computer containing a 68K central processing unit that runs Macintosh system software. Compare **PowerPC-based Macintosh computer.**

alignment convention The way internal members of an aggregate type are laid out in memory. The convention affects the size of the aggregate type and the offset of each member from the start of the aggregate type.

ANSI C language dialect The C programming language dialect that adheres to the language defined by the ANSI document *American National Standard Institute* (ANSI X3.159-1991). See also **strict ANSI C.**

ANSI C library A collection of functions that support code written in ANSI C or C++ for the PowerPC environment. The library provides the entire C library defined by the ANSI C standard as well as support for Apple extensions (such as Pascal string functions).

C++ inline function Any C++ function specified with the `inline` keyword or defined within a class definition.

C++ language dialect The C++ programming language dialect that adheres to the de facto C++ standard except for templates and exception handling. See also **de facto C++ standard** and **strict C++.**

C++-style comment A source code comment that begins with the delimiter `//` as defined by the de facto C++ standard.

CFM-68K runtime architecture A 68K Macintosh runtime architecture that uses the Code Fragment Manager. Its handling of fragments and the ability to use shared libraries is analogous to that of the PowerPC runtime architecture, but it differs in a number of details because of system limitations. In particular, it uses segmented

application code addressed through a jump table. See also **classic 68K runtime architecture.**

classic 68K runtime architecture The runtime architecture that has been used historically for the 68K Macintosh. Its defining characteristics are the A5 world, segmented applications addressed through the jump table, and the application heap for dynamic storage allocation. See also **CFM-68K runtime architecture.**

constant folding An operation that takes a constant expression and turns it into a single constant.

contraction operation The combination of a multiplication operation with either an addition or a subtraction operation, which results from a multiply-add instruction. See also **multiply-add instruction.**

de facto C++ standard The current C++ language definition described in the ANSI working paper *American National Standard Institute* (ANSI X3J16). This definition is based on the CFront version 3.0 from USL (UNIX System Laboratories).

error message An SC diagnostic message indicating an error condition that must be corrected for compilation to continue.

export list A list of exported functions and objects, defined within a compilation unit, that have external linkage. See also **export list file.**

export list file A file that contains an export list. See also **export list.**

fat application An application that contains code of two or more instruction sets. See also **68K application.**

header file A file that you merge into your source file using the `#include` preprocessor directive.

host computer The Macintosh computer on which the SC compiler resides.

implementation-specific detail Certain compilation or language features that the ANSI C language definition leaves to the discretion of the implementor.

import library A library containing functions that are not copied directly into your program at link time but are instead referenced by your program when it runs on the target system.

include file See **header file.**

inlining candidate One of three types of functions that qualifies for inlining by SC but that the compiler expands only if it determines a resulting speed benefit without excessive code expansion.

Interface library A collection of header files and functions to support the Macintosh Toolbox and provide compatibility with existing system extensions and PowerPC applications.

intrinsic ANSI C library function A library function assumed by SC to be known because it is specified in the ANSI C language definition and brought into scope by the inclusion of a standard ANSI C library header file such as `stdio.h`.

leaf function A function that contains no function calls.

local optimization A type of optimization performed on extended basic blocks, which includes constant folding,

constant propagation, expression transformations and strength reductions, copy propagation, common subexpression elimination, and dead code elimination. A local optimization does not increase code size and, when possible, may reduce it.

multiply-add instruction A floating-point instruction that combines a multiplication operation with either an addition or a subtraction operation into a single operation. See also **contraction operation.**

no optimization The suppression of all SC optimization. See also **local optimization, size optimization,** and **speed optimization.**

Numerics library A collection of functions, macros, and type definitions that support operations on floating-point values.

object file A file that contains relocatable code and results from compilation of a source code.

Pascal string A sequence of characters that begins with a length byte, uses `\p` as its first character, and has a maximum size of 255 characters.

PEF See **Preferred Executable Format.**

PowerPC alignment An alignment convention supported by SC that aligns each element to provide the fastest memory access on the PowerPC architecture. See also **68K alignment** and **alignment convention.**

PowerPC-based Macintosh computer Any computer containing a PowerPC central processing unit that runs Macintosh system software. Compare **68K-based Macintosh computer.**

preferred alignment A laying out of the elements in a data structure in memory, loosely based on the byte boundary that provides the most efficient access for the given type.

pragma An ANSI C language standard feature that lets you include in your source files directives to a specific compiler while still producing ANSI-compliant code.

Preferred Executable Format (PEF) The format of executable files used for PowerPC applications.

SC compiler An ANSI-compliant C compiler that produces optimized code for the 68K environment.

SC or C++ library A collection of functions that support code written in C++ for the 68K environment. The library contains two parts, Stream and Support, and provides functions to which SC can generate direct calls.

SCpp compiler An ANSI-compliant C++ compiler that produces optimized code for the 68K environment.

size optimization A type of optimization that reduces the size of your code, even if the result is a decrease in speed.

source code dialect Any specific version of a programming language. See also **ANSI C language dialect** and **C++ language dialect.**

source file A text file that contains C or C++ source code.

Standard C runtime library An object file that provides functions that are required to support an ANSI C execution environment.

speed optimization An optimization that provides the maximum optimization level for highest speed performance. Speed optimizations include all local optimizations, as well as global constant propagation, code hoisting, loop unrolling, induction variable elimination on inner loops, nested loops, and entire functions (that is, the elimination of partial and total redundancies), redundant storage elimination, global copy propagation, and inlining of user functions (including C functions). See also **local optimization.**

strict ANSI C The SC implementation of the C language dialect that strictly conforms to the ANSI definition and that is controlled by the `-ansi strict` option. See also **ANSI C language dialect.**

strict C++ The SC implementation of the C++ language dialect that strictly conforms to the de facto C++ standard and that is controlled by the `-ansi strict` option. See also **C++ language dialect.**

target computer The PowerPC-based Macintosh computer on which you run code compiled by SC.

trigraphs Sequences of three letters that are treated as one by the compiler.

user-defined function Any C or C++ function whose definition is visible to SC from a call site.

warning message A compiler diagnostic message indicating usage in your source code that requires your attention but that does not terminate compilation.

Index

Symbols

Numerals

A

B

C

D

E

F

H

I

J

K

L

M

N

O

P

Q, R

S

T

U

V

W

X, Y, Z

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and FrameMaker software. Proof pages were created on an Apple LaserWriter Pro printer. Final pages were created on a Docutek. Line art was created using Adobe™ Illustrator and Adobe Photoshop. PostScript™, the page-description language for the LaserWriter, was developed by Adobe Systems Incorporated.

Text type is Palatino® and display type is Helvetica®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Adobe Letter Gothic.

WRITER
Jim Lawrie