MACINTOSH
PROGRAMMER'S
WORKSHOP

MrC/MrCpp

C/C++ Compiler for Power Macintosh

Version 1.0

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, *MrC/MrCpp: C/C++ Compiler for Power Macintosh*, is a reference for MrC and MrCpp, which are ANSI-compliant C and C++ compilers that produce highly optimized code for the PowerPC™ environment. It describes the use and features of the MrC and MrCpp compilers.

The MrC and MrCpp compilers run within MPW on both the Power Macintosh and 680x0 families of computers, but generate code only for the PowerPC environment. They are part of a tool suite for the PowerPC environment that includes a debugger, an assembler, and a linker. You can use MrC and MrCpp to recompile existing 680x0 source code for the PowerPC environment. You can also use them in combination with 680x0 compilers (such as SC and SCpp) to create applications that contain both PowerPC and 680x0 code. You cannot use MrC or MrCpp to generate 680x0 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)

You also need to be familiar with the PowerPC runtime environment. See *Inside Macintosh: PowerPC System Software* for a detailed description.

This book contains the following three chapters:

- Chapter 1, “About MrC,” provides an overview of the features of MrC and MrCpp, including the language dialects and extensions they support, certain implementation-defined details, and the accompanying libraries.
- Chapter 2, “Using MrC,” 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, “MrC Reference,” gives a detailed description of each compiler option and pragma and describes the predefined symbols.

Related Documentation

This book is part of Apple's documentation suite for the PowerPC environment. This suite includes *Inside Macintosh: PowerPC System Software*, *Inside Macintosh: PowerPC Numerics*, *Assembler for Macintosh With PowerPC*, and the *Macintosh Debugger Reference*, second edition. Where appropriate, this book points you to pertinent parts of those books.

For information on using MPW, see the books *Introduction to MPW,* second edition; *Building and Managing Programs in MPW,* second edition; and *MPW Command Reference*.

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 PowerPC ANSI C library (StdCLib) and MrC C++ library (MrCPlusLib). 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 *American National Standard for Information Systems—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 PowerPC ANSI C library, see *Building and Managing Programs in MPW*, second edition.

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, and the 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 notation to describe compiler commands:

[ ]	Brackets indicated that the enclosed elements are optional.
italic	Italic font indicates that you must substitute a real value that matches the definition of the element.
...	Elipses (...) 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

APDA is Apple's worldwide source for hundreds of development tools, technical resources, training products, and information for anyone interested in

developing applications on Apple platforms. Customers receive the *APDA Tools Catalog* featuring all current versions of Apple development tools and the most popular third-party development tools. APDA offers convenient payment and shipping options, including site licensing.

To order products or to request a complimentary copy of the *APDA Tools Catalog*, contact

APDA
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	APDA
America Online	APDAorder
CompuServe	76666,2405
Internet	APDA@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

About MrC

Contents

This chapter provides a general introduction to the MrC and MrCpp compilers, which are C and C++ compilers, respectively. In this book, *MrC* is used to refer to both compilers unless there is a reason to distinguish between them.

Introducing the MrC Compiler

The MrC compiler runs on both Power Macintosh and 680x0 Macintosh computers and produces code for the PowerPC™ runtime architecture. The MrC 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 680x0 code with MrC. You can, however, create **fat applications** using MrC together with a 680x0 compiler (such as SC or SCpp), and Rez, the MPW resource compiler. A fat application is one that contains both PowerPC and 680x0 code. See *Building and Managing Programs in MPW*, second edition, for more information.

The computer you use to compile the code (the **host computer**) can be either a 680x0 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 MrC 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 Power 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 MrC 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, which is the language defined by the American National Standards Institute (ANSI) in the document *American National Standard for Information Systems—Programming Language—C* (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 for Information Systems—Programming Language—C++* (ANSI X3J16), which is based on CFront version 3.0 from USL (UNIX System Laboratories).

You use the `MrC` command to compile C source code files, and the `MrCpp` 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. To ensure compatibility with such tools you may want to use the filename suffix `.c` for your C source files and `.cp` for C++ source files.

Strict C and C++ Language Standards

MrC and MrCpp do not conform strictly to the ANSI C and C++ language standards when compiling unless you request it. This means that, by default, MrC 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," beginning on page 1-8); `-ansi strict` does not allow you to use Apple extensions. In either case, MrC issues error messages rather than warning messages for any nonconforming usages. 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 "MrC Messages," beginning on page 2-11.)

Most of the header files supplied with MrC require using the `-ansi off` option (the default setting for the MrC 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 MrC 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," beginning on page 3-5, for more information about language-specific options.

Libraries

Apple has developed six libraries to support programming for the PowerPC runtime architecture. These include the standard C and C++ libraries (with Apple extensions), the PowerPC Numerics and interface libraries, and the standard C and C++ runtime libraries.

The ANSI C, Numerics, and interface libraries are import libraries, a type of library that might be new to you. With an **import library,** the library code is not copied directly into your program at link time. Instead, your program references the library code when it runs on the target system. As a result, users can simultaneously run several applications that access the same code in an import library. Unlike individual libraries, however, the import library your program uses must be installed on the target system. Power Macintosh computers come with the PowerPC ANSI C, Numerics, and interface libraries already installed. The remaining PowerPC libraries are all individual libraries. For more information on import libraries, see *Inside Macintosh: PowerPC System Software*.

Note
The MrC import libraries contain PowerPC code and are therefore not suitable for use with 680x0 applications. ◆

Here is a description of each library:

The **PowerPC ANSI C library** (StdCLib) provides functions to support code written in ANSI C or C++. The library contains the entire C library defined by the ANSI C specification, which provides constants, data types, and functions that support formatted and low-level input/output, string manipulation, character classification, memory allocation, and integer mathematical

functions. It also provides support for Apple extensions (such as Pascal string functions).

The **MrC C++ library** (MrCPlusLib) provides functions to support code written in C++. The library contains two parts, Stream and Support. Stream is based on the USL (UNIX System Laboratories) 3.0 C++ Stream library and is expected to be part of the eventual ANSI C++ language standard. Stream provides a set of classes and routines for C++ input and output. Support was developed by Apple and provides functions to which MrC can generate direct calls. One of the Support functions is `__cplusstart`, which is the entry point for a typical C++ application. In addition, the function `__cplusstart` ensures the initialization of global and static objects that require runtime initialization and sets up exit code for the destruction of global and static objects with destructors. The function `__cplusstart` then calls your program's function `main`.

The **PowerPC Numerics library** (MathLib) provides function definitions to support operations on floating-point values.

The **PowerPC interface library** (InterfaceLib) provides a set of functions to support the Macintosh Toolbox and to provide compatibility with existing system extensions and PowerPC applications. As a result, calls between PowerPC and 680x0 code are transparent.

The **PowerPC standard C runtime library** (StdCRuntime.o) provides functions that are required to support an ANSI C execution environment.

The **PowerPC C runtime library** (PPCCRuntime.o) contains PowerPC-specific runtime support functions. MrC generates calls to these functions. These functions are used for converting between floating-point and integer data types, setting up global links for calls through function pointers, and providing low-level functions that support the 128-bit format `long double` type.

For more information, see the discussion of the PowerPC ANSI C library extensions in *Building and Managing Programs in MPW*, second edition.

Object File Formats

MrC produces object files in the IBM AIX 3.2 **Extended Common Object File Format (XCOFF).** Although XCOFF is an executable format, Apple has developed a more efficient format, the **Preferred Executable Format (PEF).** You should therefore convert your linked XCOFF object files into PEF files before

executing them. You can create PEF files directly from the PPCLink linker or by using the MakePEF tool as described in the book *MPW Command Reference.*

Apple Language Extensions

By default, MrC 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 MPW 680x0 compilers, the Pascal calling conventions differ from C and C++ calling conventions. With the PowerPC runtime architecture, however, these calling conventions are identical. The MrC compiler therefore 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, 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 MrC 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 MrC 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

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

Unsupported Apple Language Extensions

The MrC compiler does not support the SANE `extended` or `comp` keywords. You should use the `long double` data type instead. MrC supports 64-bit and 128-bit `long double` data types. You use the `-ldsize` option, described on page 3-10, to specify the size of the `long double` data type.

Note
Using the `long double` data type wherever possible can significantly increase the portability of your code. If you use the `long double` data type and do not use any other Apple language extensions in your source code, you can compile your code in strict ANSI C. Furthermore, the `long double` data type is equivalent to the SANE `extended` data type in the 680x0 runtime architecture. ◆

CHAPTER 2

Using MrC

Contents

This chapter shows you how to use the MrC 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*, second edition, and *Macintosh Debugger Reference*, second edition, 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 MrC compiler typically uses the following syntax, which includes the MrC invocation, options, and source filenames:

`MrC` [*option list*] *filename*

Invoking MrCpp is similar:

`MrCpp` [*option list*] *filename*

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

```
MrC -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 MrC command instead of the MrCpp command indicates that you want to compile in ANSI C.

To compile a C++ file, you use MrCpp:

```
MrCpp -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:

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

Note
The `-sym` option is not compatible with any MrC optimizations and automatically suppresses their use. ◆

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

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

Table 2-1 Alphabetical list of MrC command line options

Option	Purpose
`-align` *parameter*	Aligns data structures according to either PowerPC or 680x0 alignment rules (page 3-5).
`-ansi` *parameter*	Sets the level of ANSI language conformance (page 3-14).
`-c`	Checks the syntax of the code in the source file without generating an object file (page 3-29).
`-char` *parameter*	Determines the implementation of the `char` data type (page 3-7).
`-curdir`	Sets the directory for temporary files to the current working directory (page 3-40).
`-d` *parameter*	Defines a preprocessor symbol name (page 3-16).
`-dump` *filename*	Saves the state of the compilation in a file (page 3-41).
`-e`	Generates preprocessor output (page 3-29).

Table 2-1 Alphabetical list of MrC command line options (continued)

Option	Purpose
`-enum` *parameter*	Controls the size of enumeration types (page 3-8).
`-export_list` *filename*	Changes the name of the export list file (page 3-38).
`-fp_contract` *parameter*	Controls whether MrC generates PowerPC multiply-add instructions (page 3-20). (This option is identical to the `-maf` option.)
`-i` *pathname[,pathname...]*	Adds a directory to the list of header file directories (page 3-43).
`-inclpath` *parameter*	Controls how header files are looked up (page 3-44).
`-inline` *parameter*	Controls the type of inlining performed by MrC (page 3-23).
`-j0`	Recognizes 2-byte sequences for Japanese characters (page 3-12).
`-j1`	Recognizes 2-byte sequences for Taiwanese or Chinese characters (page 3-13).
`-j2`	Recognizes 2-byte sequences for Korean characters (page 3-13).
`-l` *filename*	Generates a listing file (page 3-30).
`-ldsize` *parameter*	Controls the size of the `long double` type (page 3-10).
`-load` *filename*	Loads a saved state of compilation (page 3-42).
`-maf` *parameter*	Controls whether MrC generates PowerPC multiply-add instructions (page 3-23). (This option is identical to the `-fp_contract` option.)
`-noMapCR`	Changes the mapping of the carriage return and line-feed characters (page 3-15).
`-notOnce`	Allows MrC to read the same header file more than once (page 3-45).

Table 2-1 Alphabetical list of MrC command line options (continued)

Option	Purpose
`-o` *filename*	Overrides the default naming conventions and directory location for generated object files (page 3-32).
`-opt` *parameter[,modifier...]*	Sets the optimization level (page 3-26). See also "Optimizations," beginning on page 2-14.
`-p`	Generates progress information about the compilation (page 3-31).
`-proto` *parameter*	Sets whether or not MrC requires function prototypes (page 3-17).
`-shared_lib_export` *parameter*	Controls whether MrC exports names of functions and objects with external linkage (page 3-37).
`-sym` *parameter[,modifier...]*	Controls whether MrC generates debugging information for your source file (page 3-33).
`-target` *parameter*	Specifies the architecture you want MrC to compile for (page 3-27).
`-typecheck` *parameter*	Controls whether MrC performs strict or relaxed typechecking (page 3-18).
`-w` *parameter*	Controls how warning messages are issued (page 3-34). See also "MrC Messages," beginning on page 2-11.
`-x`	Allows more than the normal maximum number of errors without terminating the compilation (page 3-36).
`-xa` *parameter*	Determines how accessible generated template functions are (page 3-19).
`-xi` *template*	Instantiates a specific template (page 3-18).
`-y` *directory*	Specifies where you want MrC to put temporary files (page 3-39).

Handling Files

When you compile with MrC, you generally work with four different types of files: source, object, header, and export list.

- **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.
- **Export list files**: Compiler-generated files that contain an **export list**, which has the names of functions and objects with external linkage, defined within a compilation unit.

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 MrC names files for you.

There are no restrictions for naming object and export list files. MrC accepts any filename prefix and suffix 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 MrC and MrCpp allow any suffix, other tools require `.c` for C files and `.cp` for C++ files. See the section “Language Support” on page 1-4 for more information.

MrC assigns filenames only if you do not name an object file or an export list file on the MrC command line. In the first case, if you do not name an object file on the MrC command line with the `-o` option, MrC 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`). In the second case, if you do not name an export list file on the MrC command line with the `-export_list` option and you use the `-shared_lib_export on` option, MrC uses the source

filename with a `.x` extension as the default export list filename (for example, the source file `myfile.c` results in an export file named `myfile.c.x`).

Including Header Files

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

- Quotation marks, or double quotes ("), enclosing a header filename indicate that MrC 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 `{PPCCIncludes}`, which is typically `{MPW}Interfaces:PPCCIncludes`. Use double quotes to include your own header files.
- Angle brackets (< and >) enclosing a header filename indicate that MrC should first search the directories named in the `-i` option and then the `{PPCCIncludes}` directory. Use brackets to include system header files.
- A colon (:) within a header filename indicates an explicit pathname. MrC 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, MrC 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, MrC 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 MrC 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 MrC compiler does not open a header file more than once, unless you specifically direct it to do so by using the `-notOnce` option, which is described on page 3-45.

If you use the `-notOnce` option to force MrC 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 (see page 3-50).

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 MrC compiler provides a choice of two alignment conventions for data structures, PowerPC alignment and 680x0 alignment. With **PowerPC alignment,** each element is aligned to provide the fastest memory access in the PowerPC runtime architecture. With **680x0 alignment,** MrC matches, bit for bit, the alignment used by the MPW C and MPW C++ compilers in the 680x0 runtime architecture.

By default, MrC uses PowerPC alignment conventions. You can change the default alignment convention with the `-align` option, which is described on page 3-5. You can also use the `options align` pragma, which is described on page 3-46, 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 680x0 alignment. The rules for PowerPC and 680x0 alignment are described in the book *Inside Macintosh: PowerPC System Software*.

Table 2-2 Size and preferred alignment of data types

	PowerPC		680x0	
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 on MC68881)	4
Enumeration	Varies	The preferred alignment of the same sized integral type	Varies	The preferred alignment of the same sized integral type

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

	PowerPC		680x0	
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

MrC Messages

MrC can produce warning and error messages, both of which are sent to

standard error, the file in which MrC 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 read about 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 on 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 MrC compiler repeats the line of code for each of the messages.

Warning Messages

MrC 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. Warning messages do 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, which is described on page 3-34. ◆

Error Messages

MrC issues **error messages** to indicate error conditions that you must correct to compile your source code successfully. If MrC encounters an error condition 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 following the `#endif` preprocessor directive. If you do not follow this rule and compile your source code using strict ANSI C, you receive an error like this one:

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

Note
By default, MrC allows four compiler errors before it terminates compilation.You can tell MrC to report all error messages, no matter how many there are, by using the `-x` option, which is described on page 3-36. ◆

MrC 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 which warnings MrC produces, and set the location where MrC puts generated object files.

- To compile code through only 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 check the syntax of your code without compiling, use the `-c` option.
- To generate progress information about the compilation session, use the `-p` 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 MrC issues, use the `-x` option.
- To control the type of debugging information MrC produces, use the `-sym` option.
- To send the names of functions and objects with external linkage to a file, use the `-shared_lib_export` option. Use the `-export_list` option to specify the name of the export file (if you want to use a name other than the default).

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

Optimizations

This section describes the MrC optimizations and how to best select optimizations for your particular source code. MrC supports the use of local optimizations, speed optimizations, size optimizations, or no optimizations. By default, MrC performs only local optimizations. You can control the optimization level, however, with the `-opt` option, which is described on

page 3-26. The level of optimization you select also affects the type of inlining performed by MrC. See the description of the `-inline` option on page 3-23 for more information.

Local optimizations (`-opt local`) improves code performance using only information that is local to extended basic blocks. These optimizations include elimination of dead code and **constant folding,** which involves taking a constant expression and turning it into a single constant. In addition to local optimizations, `-opt local` also performs global register allocation.

Speed optimizations (`-opt speed`) provide the maximum optimization level for the highest code performance and include all local optimizations (`-opt local`), as well as inlining, instruction scheduling, and loop unrolling.

Size optimizations (`-opt size`) reduce the size of your code, even if the result is a decrease in speed.

No optimizations (`-opt off`) suppress the use of all MrC optimizations.

Note
Using the `-sym on` option (for debugging) automatically suppresses all optimizations (`-opt off`). For a description of the `-sym` option, see page 3-33. ◆

The next few sections show the effects of the different optimization levels using the `example.c` file, which looks like this:

```
#include <stdio.h>

int gus (int y)
{
    if (y == 1) {
        return 5;
    } else if (y == 3) {
        return 7;
    } else {
        return 9;
    }
}
int main (void)
{
    int local = 3;
```

```
        printf("The value is '%d'\n", gus(local));
        return 0;
}
```

As you review these examples, keep in mind that the way in which optimization levels affect your code size and speed depends on the nature of your source code. Therefore, the effect of the optimizations that you have chosen might vary from file to file, or even from function to function.

For example, you might use inlining functions to decrease execution time by avoiding the expense of a call. If you use a function many times, you might expect that inlining would increase your code size. Inlining a function, however, can allow so many more optimizations to occur at the point of the call that the size of the code can actually decrease. (For instance, compare the size of the code example compiled with local optimizations on page 2-18 with the code example compiled with speed optimizations on page 2-19.) Alternatively, the additional code can cause the function to take longer to load or result in misses in the instruction cache. Both of these situations can actually slow execution time.

Similarly, loop unrolling normally decreases the execution time of a loop while increasing its size. If you have already fine-tuned and unrolled the loop at the source level, however, there might be only small additional savings. As with inlining functions, unrolling loops can also cause instruction cache misses, thus actually slowing the code.

For best results, experiment with all levels of optimization and function inlining to find the best balance of speed and size optimizations for each application.

Compiling With No Optimizations

The following command compiles `example.c` using no optimizations:

```
MrC -opt off example.c
```

Listing 2-1 shows the resulting generated code.

Listing 2-1 Code generated from compiling with no optimizations

```
Loc    Object Code  Label      Opcode    Operand                    Comment
0000                           csect     .gus{PR}                   ; [10]
0000    90610018               stw       r3,0x0018(SP)
0004    80610018               lwz       r3,0x0018(SP)
0008    2C030001               cmpwi     r3,1
000C    40820010               bne       $+0x0010                   ; 0x0000001C
0010    38600005               li        r3,5
0014    4E800020               blr
0018    48000020               b         $+0x0020                   ; 0x00000038
001C    80810018               lwz       r4,0x0018(SP)
0020    2C840003               cmpwi     cr1,r4,3
0024    40860010               bne       cr1,$+0x0010               ; 0x00000034
0028    38600007               li        r3,7
002C    4E800020               blr
0030    48000008               b         $+0x0008                   ; 0x00000038
0034    38600009               li        r3,9
0038    4E800020               blr

003C                           csect     .main{PR}                  ; [13]
003C    7C0802A6               mflr      r0                         ; LR = 8
0040    90010008               stw       r0,0x0008(SP)
0044    9421FFC0               stwu      SP,-0x0040(SP)
0048    38600003               li        r3,3
004C    90610038               stw       r3,0x0038(SP)
0050    80610038               lwz       r3,0x0038(SP)
0054    4BFFFFAD               bl        .gus{PR}                   ; 0x00000000 {0}
0058    60000000               nop
005C    60640000               ori       r4,r3,0x0000
0060    80620000               lwz       r3,.stringBase0{TC}(RTOC) ; 0x00000080 {1}
0064    4BFFFF9D               bl        .printf                    ; 0x00000000 {2}
0068    60000000               nop
006C    38600000               li        r3,0
0070    80010048               lwz       r0,0x0048(SP)
0074    30210040               addic     SP,SP,64
0078    7C0803A6               mtlr      r0                         ; LR = 8
007C    4E800020               blr
```

With no optimizations, MrC generates instructions in a highly literal fashion. Every variable has a memory location and each is treated as a `volatile`

variable; every access is loaded from the memory location and every assignment updates the memory location. For example, in the function `gus`, the parameter `y` is loaded from the stack each time it is needed, as shown in lines 0004 and 001C. Also note that lines 004C and 0050 show a store and load of the `local` variable. Even variables declared as `register` receive a stack location.

Additionally, all language constructs are fully expanded. For instance, there are two returns (the `blr` instructions at lines 002C and 0038) at the end of the function `gus`. The first instruction is the return of the expression `return 9`. The second instruction is the default return that is always present at the end of a function. At lines 0014 and 0018, a return instruction is followed immediately by a branch, which is the end of block for the `if` clause. Note that even if certain instructions might be unreachable, they are generated if the language requires them.

Compiling With Local Optimizations

Compiling with local optimizations is the MrC compiler's default behavior.

The following command compiles `example.c` using only local optimizations:

```
MrC -opt local example.c
```

Listing 2-2 shows the resulting generated code.

Listing 2-2 Code generated from compiling with local optimizations

```
Loc    Object Code  Label      Opcode    Operand                 Comment
0000                           csect     .gus{PR}                ; [10]
0000    2C030001               cmpwi     r3,1
0004    4082000C               bne       $+0x000C                ; 0x00000010
0008    38600005               li        r3,5
000C    4E800020               blr
0010    2C830003               cmpwi     cr1,r3,3
0014    4086000C               bne       cr1,$+0x000C            ; 0x00000020
0018    38600007               li        r3,7
001C    4E800020               blr
0020    38600009               li        r3,9
0024    4E800020               blr
```

```
0028                        csect     .main{PR}                ; [13]
0028    7C0802A6            mflr      r0                       ; LR = 8
002C    90010008            stw       r0,0x0008(SP)
0030    9421FFC8            stwu      SP,-0x0038(SP)
0034    38600003            li        r3,3
0038    4BFFFFC9            bl        .gus{PR}                 ; 0x00000000 {0}
003C    60000000            nop
0040    60640000            ori       r4,r3,0x0000
0044    80620000            lwz       r3,.stringBase0{TC}(RTOC) ; 0x00000068 {1}
0048    4BFFFFB9            bl        .printf                  ; 0x00000000 {2}
004C    60000000            nop
0050    38600000            li        r3,0
0054    80010040            lwz       r0,0x0040(SP)
0058    30210038            addic     SP,SP,56
005C    7C0803A6            mtlr      r0                       ; LR = 8
0060    4E800020            blr
0064    60000000            dc.l      0x60000000               ; csect pad
```

With local optimizations specified, MrC eliminates all the redundant store and load operations. For instance, the variable `local` is now in register R3 (see line 0034). The parameter `y` is also in register R3. All the redundant branches and returns are gone, and there is only a single return or epilog at the end of each function.

Compiling With Speed Optimizations

The following command compiles `example.c` using speed optimizations:

```
MrC -opt speed example.c
```

Listing 2-3 shows the resulting generated code.

Listing 2-3 Code generated from compiling with speed optimizations

```
Loc   Object Code  Label      Opcode    Operand                  Comment
0000                          csect     .gus{PR}                 ; [10]
0000    2C030001              cmpwi     r3,1
0004    4082000C              bne       $+0x000C                 ; 0x00000010
0008    38600005              li        r3,5
```

```
000C   4E800020              blr
0010   2C030003              cmpwi     r3,3
0014   4082000C              bne       $+0x000C                  ; 0x00000020
0018   38600007              li        r3,7
001C   4E800020              blr
0020   38600009              li        r3,9
0024   4E800020              blr

0028                         csect     .main{PR}                 ; [13]
0028   7C0802A6              mflr      r0                        ; LR = 8
002C   90010008              stw       r0,0x0008(SP)
0030   9421FFC8              stwu      SP,-0x0038(SP)
0034   80620000              lwz       r3,.stringBase0{TC}(RTOC) ; 0x00000058 {0}
0038   38800007              li        r4,7
003C   4BFFFFC5              bl        .printf                   ; 0x00000000 {1}
0040   60000000              nop
0044   38600000              li        r3,0
0048   80010040              lwz       r0,0x0040(SP)
004C   30210038              addic     SP,SP,56
0050   7C0803A6              mtlr      r0                        ; LR = 8
0054   4E800020              blr
```

With the function `gus`, the code now uses only the CR0 Condition Register field, instead of both the CR0 and CR1 Condition Register fields. (In a more complex function, this reduction in register usage might allow expression results to be saved in condition registers, thus saving recalculation.)

With speed optimizations, the benefits of inlining C functions are apparent. You see the most significant speed savings in the function `main`. Because the function `gus` is relatively simple, it is inlined in place of the call. Because the parameter is now a known constant, all the conditional expressions can be evaluated at compile time. As a result, the call to function `gus` is reduced to a single load of the constant `7` as the second parameter to the function `printf`. If the function `gus` were a static function, the code for it would not even be put in the XCOFF output, because the only reference to it no longer exists.

With speed optimizations, small accessor functions that are visible at compilation time (like those in C++) are as efficient as macros at runtime. In addition, you gain the benefits of compile-time parameter checking.

Optimizations and Function Overriding

You should not use certain optimizations if you want to override the implementation of an individual function and provide a new implementation. At link time, you can provide the new implementation on the linker command line prior to the original implementation. At runtime, you can override a function exported from a shared library by providing the new implementation in a shared library update file.

Certain optimizations, because they rely on the implementation of locally defined functions, are incompatible with function overriding. One example of this incompatibility is the inlining of functions. In the code example compiled with speed optimizations on page 2-19, the implementation of the function `gus` allows the optimizer to entirely remove the call to that function. The function `gus` is not called at runtime, so you cannot override its original implementation. A more subtle example shows up when you use a function that contains no function calls, that is, a **leaf function.** Global register allocation monitors which registers are used by a leaf function. As a result, global registers might not be saved around a call to such a function, because it is known that the called leaf function does not use those registers.

The optimizations performed by the `-opt local` option do not depend upon the implementation of any function other than the one being compiled and can therefore be safely used with function overriding. To override a function at link time, use only the `-opt local` or `-opt off` options and suppress all inlining (`-inline none`). To override a function exported from a shared library, suppress all inlining and use the `-shared_lib_export on` option, which suppresses optimizations that are incompatible with function overriding.

MrC Reference

Contents

This chapter provides a detailed reference for the MrC 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.

MrC Options

MrC 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 MrC command line:

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

This command line specifies that MrC 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, if you use the `-align power` command line option when compiling, and the `pragma options align=mac68k` pragma appears in one source file in order to set the alignment of one particular data structure, the `align` pragma takes precedence.

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

For each option description, the reference 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 just a few 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 MrC 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 not. You can use the `-enum` option to determine the size of the `enum` type. The `-ldsize` option allows you to change the length of the type referred to by the `long double` type name.

-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 680x0 alignment rules.

DESCRIPTION

The `-align` option allows you to choose whether the MrC compiler uses the PowerPC or 680x0 alignment rules for data structures as the default setting. For

optimum performance in the PowerPC runtime architecture, use PowerPC alignment. For compatibility with the 680x0 runtime architecture, use 680x0 alignment.

For more information about what these alignment rules are, see "Data Structure Management," beginning on 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 power
```

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, MrC 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 680x0 alignment:

```
-align mac68K
```

With 680x0 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 680x0 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 MrC 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. Note that 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, suppose you want to convert a `char` variable with a value of 0xFF into a `short` variable. If the `char` type is unsigned, the short 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 1, 2, or 4 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 */
```

-ldsize

You can use the `-ldsize` option to control whether the `long double` type uses a 128-bit format or a 64-bit format.

`-ldsize` *parameter*

The available values for *parameter* are

`128`	The `long double` data type is 128 bits.
`64`	The `long double` data type is 64 bits, the same as the `double` data type.

DESCRIPTION

The `-ldsize` option determines whether the `long double` type uses a 128-bit format, which is a format not native to the PowerPC runtime architecture, or a 64-bit format, which is the same format used by the `double` type and is native.

The **128-bit format** is made up of two double-format numbers. This non-IEEE format has the same range as the `double` type, but offers much more precision. Apple recommends that you use the 128-bit format only for internal data that require extra precision. You should be aware of several limitations of the 128-bit format.

1. Computations that use 128-bit format are significantly slower than computations that use the `double` or `float` types.
2. Because 128-bit format has an indefinite precision (unlike the `float` and `double` types), you cannot assume a binary representation of a fixed length.

The 128-bit format is therefore not recommended for formatted input or output of decimal data. You can use the 128-bit format, however, for binary input or output of the 128-bit `long double` values. For more information on the 128-bit `long double` format and for limitations on converting it to or from a decimal format, see *Inside Macintosh: PowerPC Numerics*, which refers to the 128-bit `long double` format as the `double double` type.

Like the `double` type, the **64-bit format** stores floating-point values of up to 15- or 16-decimal digit precision. When the `long double` size is set to 64 bits (`-ldsize 64`), the `long double` declaration and the `double` declaration are considered identical by the MrC compiler. In this case, you no longer get the precision advantage of the `long double` type. However, using 64-bit `long`

`double` types may be useful if `long double` variables or declarations occur in your source code but you want to avoid 128-bit `long double` arithmetic.

The following types of problems can occur for certain scanning, printing, and floating-point library functions when you use the 64-bit `long double` type:

- The formatted print functions `fprintf`, `printf`, `sprintf`, `vfprintf`, `vprintf`, and `vsprintf` print the wrong value for an argument of the 64-bit `long double` type and for any subsequent argument following its use. To avoid this problem, use a format item that specifies a `double` type rather than a `long double` type. The same problem occurs with the C++ output stream operator (<<) and is unavoidable.
- The formatted scanning functions `fscanf`, `scanf`, and `sscanf` result in random crashes of both your application and system if you try to read a value into an argument of the 64-bit `long double` type. These functions write 128 bits of data to the space pointed to by the corresponding argument address. Because a 64-bit `long double` variable does not have enough space to receive the data, the 64 bits following the `long double` variable in memory are overwritten. If an argument of the `long double` type is stored on the stack, the stack is also overwritten and randomly damaged. You can avoid this problem by using a format item that specifies a `double` type rather than a `long double` type. The same problem occurs with the C++ input stream operator (>>) and is unavoidable.
- The function `long double modfl(long double _x, long double *_ip)` returns an incorrect, random result and results in random crashes of both your application and system. To avoid this problem, do not use the function `modfl`. Instead, use the function `modf`.
- The single-argument floating-point functions—that is, the 58 functions defined in the file `fp.h` of the form `long double <cosl>(long double x)`—result in random crashes of both your application and system. You cannot avoid this problem unless you change your source file to call the double precision functions.
- The function `void x80told(extended *x80, long double *x)` results in random crashes of both your application and system.

Note
Mixing 128-bit and 64-bit `long double` formats in the same program can result in problems when data containing the `long double` type is shared between modules. These problems are similar to those that occur when you mix the 80-bit and 96-bit `extended` types in the 680x0 runtime architecture. ◆

DEFAULT

```
-ldsize 128
```

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 MrC 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 MrC compiler does not recognize 2-byte Asian characters.

SEE ALSO

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

-j1

You can use the `-j1` option to allow MrC 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 MrC 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 MrC 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 MrC compiler does not recognize 2-byte Asian characters.

SEE ALSO

See the `-j0` (page 3-12) and `-j1` (page 3-13) options.

Setting Conformance Standards

You can set the degree to which MrC 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 MrC 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 MrC 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 MrC 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 MrC compiler warns you that there is a possible conflict.

DEFAULT

```
-ansi off
```

SEE ALSO

See the section "Language Support," beginning on page 1-4.

-noMapCR

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

```
-noMapCR
```

DESCRIPTION

The `-noMapCR` option tells the MrC compiler to use the line-feed character for newline. Because Macintosh text files use the carriage return character to end lines, by default Macintosh C compilers generate a carriage return character for the `\n` newline escape sequence in character and string literals. C compilers on most other platforms, however, generate the line-feed character for the newline escape sequence.

This option also controls what is generated by the `\r` escape sequence. By default, the `\r` escape sequence generates a line-feed character in the MrC compiler; with the `-noMapCR` option the compiler generates a carriage return character, so that `\n` and `\r` never generate the same character.

You may find this option useful if you are compiling for other platforms.

DEFAULT

The default is for a carriage return to be generated for a newline escape sequence.

Using Preprocessor Symbols

You can define preprocessor symbols. The `-d` option allows you to define one preprocessor symbol.

-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 MrC 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 MrC performs.

-proto

You can use the `-proto` option to set whether or not MrC 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 MrC requires function prototypes in your source files.

If you use the `-proto auto` option, MrC 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 MrC 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 MrC 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, described on page 3-54.

-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, described on page 3-55.

Generating Code

MrC provides several options that allow you to control how code is generated. You can control inlining of functions, the level of optimization performed, or what specific member of the PowerPC family MrC should generate the code for.

The `-fp_contract` and `-maf` options, which are identical, allow you to control whether MrC generates PowerPC multiply-add instructions.

You can use the `-inline` option to control inlining of functions. The `-opt` option allows you to control the types of optimizations MrC performs. The `-target` option allows you to specify the architecture for which you want MrC to compile your application.

-fp_contract

You can use the `-fp_contract` option to control whether MrC generates PowerPC multiply-add instructions.

`-fp_contract` *parameter*

The available values for *parameter* are

`on`	Generate multiply-add instructions.
`off`	Don't generate multiply-add instructions.

DESCRIPTION

The `-fp_contract` option tells MrC whether to generate multiply-add instructions. It is the command line equivalent of the PowerPC Numerics pragma `fp_contract`, which is described in Appendix D of *Inside Macintosh: PowerPC Numerics*.

A **multiply-add instruction** is a floating-point instruction that combines a multiplication operation with either an addition or a subtraction operation. The resulting single operation, also known as a **contraction operation,** offers several advantages. First, a single multiply-add instruction can perform a combined multiplication and addition operation in the same time that a multiplication operation takes, thus providing much faster floating-point computations. Second, no more than a single rounding error occurs in the multiply-add computation, compared with the two possible rounding errors that can occur for the two replaced instructions. As a result, the computation can provide a more accurate result.

Note
For some input values, the result of a multiply-add computation is slightly different than if the multiply and add operations were performed separately. In certain programs, this difference in value might be unacceptable. ◆

If you use the `-fp_contract on` option, MrC combines appropriate sequences of floating-point multiplication and addition instructions into a single multiply-add instruction. If you use the `-fp_contract off` option, MrC does not generate any multiply-add instructions and therefore provides identical results for similar expressions.

DEFAULT

```
-fp_contract on
```

EXAMPLE

Assume the file `MAF_test.c` contains the following code:

```
double MAF_test(double x, double y, double z)
{
    return (x*y + z);
}
```

The following command tells MrC to generate multiply-add instructions when it compiles the `MAF_test.c` file:

```
MrC -opt off "MAF_test.c"
```

Here is the resulting generated code:

```
Loc   Object Code  Label      Opcode    Operand
0000                          csect     .MAF_test{PR}
0000   D8210018               stfd      fp1,0x0018(SP)
0004   D8410020               stfd      fp2,0x0020(SP)
0008   D8610028               stfd      fp3,0x0028(SP)
000C   C8010018               lfd       fp0,0x0018(SP)
0010   C8410020               lfd       fp2,0x0020(SP)
0014   C8610028               lfd       fp3,0x0028(SP)
0018   FC2018BA      ======>  fmadd     fp1,fp0,fp2,fp3
001C   4E800020               blr
```

The following command tells MrC not to generate multiply-add instructions when it compiles the `MAF_test.c` file:

```
MrC -opt off -fp_contract off "MAF_test.c"
```

Here is the resulting generated code:

```
Loc   Object Code  Label     Opcode    Operand
0000                         csect     .MAF_test{PR}
0000  D8210018               stfd      fp1,0x0018(SP)
0004  D8410020               stfd      fp2,0x0020(SP)
0008  D8610028               stfd      fp3,0x0028(SP)
000C  C8210018               lfd       fp1,0x0018(SP)
0010  C8010020               lfd       fp0,0x0020(SP)
0014  FC210032       ======> fmul      fp1,fp1,fp0
0018  C8410028               lfd       fp2,0x0028(SP)
001C  FC21102A       ======> fadd      fp1,fp1,fp2
0020  4E800020               blr
```

SEE ALSO

The `-maf` option, described next, is identical to the `-fp_contract` option.

-maf

You can use the `-maf` option to control whether MrC generates PowerPC multiply-add instructions.

`-maf` *parameter*

The available values for *parameter* are

`on`	Generate multiply-add instructions.
`off`	Don't generate multiply-add instructions.

DESCRIPTION

This option is identical to the `-fp_contract` option (page 3-20) in behavior.

DEFAULT

```
-maf on
```

-inline

You can use the `-inline` option to control the type of inlining performed by MrC.

`-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`.
value	Sets a complexity limit on the functions MrC will inline. This value must be between 0 and 5. A value of 0 is equivalent to `-inline off`, whereas a value of 5 indicates extreme complexity. A value of 2 is equivalent to `-inline on`.

DESCRIPTION

The `-inline` option lets you choose the type of inlining that MrC performs on both C and C++ functions. It applies only to functions that are inlining candidates. An **inlining candidate** is one of three types of functions that qualify for inlining by MrC but that are inlined only if MrC determines a resulting speed benefit without excessive code expansion. The functions that are inlining candidates are C++ inline functions, intrinsic ANSI C library functions, and user-defined functions.

- A **C++ inline function** is any function that you specify with the `inline` keyword or that is defined within a class definition.
- An **intrinsic ANSI C library function** is a library function assumed by MrC 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`.
- A **user-defined function** is any C or C++ function whose definition is visible to MrC from a call site.

You can specify inline expansion of no inline candidates, only certain inline candidates, or all inline candidates. Remember, however, that requesting MrC to perform inlining does not guarantee that it will expand all inlining candidates.

Note
In the PowerPC runtime architecture, inlining can result in cache misses due to code expansion and therefore might not always improve code performance. ◆

DEFAULT

The setting of the `-opt` option, which controls optimization, determines the default inlining level performed by MrC.

Optimization level	Inlining default specified
`-opt off`	Inlines only functions marked with the `inline` keyword.
`-opt size`	`-inline off`
`-opt speed`	`-inline on`
`-opt local`	Inlines only functions marked with the `inline` keyword.

IMPORTANT

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

EXAMPLE

The following source code example indicates how the different inlining levels take effect:

```
#include <stdio.h>
int inline_me() { return 0; }
inline int explicitly_inline() { return 0; }
extern int cannot_inline();

void use_inlines()
{
    int i;
    i = inline_me(); /* inlined with -inline 1 */
    i = explicitly_inline();/* inlined unless -inline off */
    i = cannot_inline(); /* no definition results in no inlining */
}
```

When inlining is off, none of the functions (`inline_me`, `explicitly_inline`, and `cannot_inline`) are inlined.

When inlining is on, only the `cannot_inline` function is not inlined. The `inline_me` and `explicitly_inline` functions are inlined.

SEE ALSO

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

-opt

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

`-opt` *parameter[,modifier…]*

The available values for *parameter* are

`off`	Perform no optimizations.
`none`	Identical to `off`.
`local`	Perform local optimizations and global register allocation.
`size`	Perform optimizations for size rather than speed.
`speed`	Perform optimizations for highest performance.

The *modifier* is used only with the `speed` parameter and can have the following values:

`nounroll`	Perform speed optimizations with no loop unrolling.
`norep`	Perform speed optimizations without repeating global propagation and redundant store elimination.
`nointer`	Perform speed optimizations without interprocedural optimizations.

DESCRIPTION

The `-opt` option specifies the desired level of optimization to be performed. You can choose from among local optimizations, speed optimizations, and size optimizations. These optimization levels are described in detail in "Optimizations," beginning on page 2-14.

MrC repeats global copy propagation and redundant store elimination for each function as long as each repetition improves the generated code, unless you use the `norep` modifier value with the `speed` parameter value.

MrC performs loop unrolling unless you use the `nounroll` modifier value with the `speed` parameter value.

MrC performs interprocedural optimizations unless you use the `nointer` modifier value with the `speed` parameter value. Interprocedural optimizations may be undesirable if you plan on replacing a function at link time or runtime.

Note
Using the `-sym on` option suppresses all optimizations. ◆

DEFAULT

If `-sym off`, then `-opt local`. If `-sym on`, then `-opt off`.

EXAMPLE

The following command specifies compiling `example.c` using no optimizations:

```
MrC -sym off -opt off example.c
```

SEE ALSO

See the section "Optimizations," beginning on page 2-14.

-target

You can use the `-target` option to specify the processor architecture you want MrC to compile for.

`-target` *parameter*

The available values for *parameter* are

`ppc`	Specifies the PowerPC architecture.
`powerpc`	Identical to `ppc`.
`601`	Specifies the 601 variant of the PowerPC architecture.
`603`	Specifies the 603 variant of the PowerPC architecture.

`power` Specifies the POWER architecture.

DESCRIPTION

The `-target` option generates code that depends on the characteristics of the target architecture that you specify. The architecture may affect which instructions are available, the optimal order of instructions in the compiled code, and instruction scheduling.

The default value, `ppc`, always indicates the current model and therefore will always be appropriate for future PowerPC generations.

The POWER processor is an ancestor of the PowerPC processor and uses a slightly different instruction set.

DEFAULT

```
-target ppc
```

Controlling Output

Several MrC options allow you to control the output you get from the compiler. You can control the output MrC gives you when it compiles your source files. You can set the warning and error levels used by MrC. You can also create export lists of the names of functions and objects with external linkage.

Controlling Compilation Output

Using the `-c` option, you can check the syntax of your code without generating an object file. The `-e` option lets you stop compilation of your files after the preprocessor phase, and the `-l` option gives you a listing of your source files, with the include files expanded.

The `-p` option allows you to get a progress report on the compilation of your source files. The `-o` option allows you to choose the name for the object file created by the compilation.

The `-sym` option lets you control whether MrC generates debugging information for your source file.

-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 3-32) 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 MrC 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 MrC, 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, page 3-29), 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 MrC 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.

-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 MrC 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` (which is described in "Optimizations," beginning on page 2-14):

```
MrC -p example.c
```

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

```
MrC -p example.c
MrC C Compiler 1.0
Copyright (C) 1994-1995 by Symantec Corporation
              1994-1995 by Apple Computer, Inc.
```

```
'example.c'
  'Donut:MPW:Interfaces:CIncludes:stdio.h'
  gus
  main
Apple PPC Code Generator, 1.0
25 instructions (104 code bytes) generated in 0 seconds
```

-o

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

`-o` *filename*

filename A filename or directory name or both.

DESCRIPTION

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

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

DEFAULT

For a filename, MrC 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 resulting object file:

```
MrC -o newname.o myfile.c
```

The command

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

sends the resulting object file to the following directory location:

```
::objects:myfile.c.o
```

-sym

You can use the `-sym` option to control whether MrC 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 MrC.

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

IMPORTANT

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

DEFAULT

```
-sym off
```

Setting Warning and Error Levels

You can control the type of warnings you receive and the number of errors MrC can give you. The `-w` option allows you to control which warnings MrC 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 MrC 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 MrC 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 MrC compiler for C language compiles are as follows:

Value	Description
`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.
`12`	A variable is used before it is set.
`13`	A floating-point constant is out of range.
`15`	The returning address of an automatic variable will be gone as soon as the function returns.
`17`	The pragma is unrecognized.
`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.

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

Value	Description
`4`	The access to the template is different from what was previously specified.
`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)`.

Value	Description
11	A nonconstant reference was initialized to a temporary variable.
14	Function definitions with separate parameter lists are obsolete in C++.
18	You are casting from an incomplete `struct` type.

DEFAULT

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

EXAMPLE

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

```
MrC -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 MrC to generate only four errors before terminating compilation.

Creating Export Lists

The `-shared_lib_export` option allows you to control whether MrC exports the names of functions and objects with external linkage, and the `-export_list` option lets you change the name of the export file.

-shared_lib_export

You can use the `-shared_lib_export` option to control whether MrC exports from an import library the names of functions and objects with external linkage, defined within the compilation unit.

`-shared_lib_export` *parameter*

The available values for *parameter* are

`off`	Do not export the code and data objects.
`on`	Export the code and data objects.

DESCRIPTION

The `-shared_lib_export` option creates an export list that can be provided to PPCLink. During linking, these names indicate which functions are to be exported from an import library. MrC will not export the names of functions and objects that you declare but do not define within the same compilation unit or that do not have external linkage (that is, static objects and functions). For more information, see the discussions of import libraries in *Inside Macintosh: PowerPC System Software* and *Building and Managing Programs in MPW*, second edition.

Note
The exported object and function names are the names used at link time. For C++, the names are therefore mangled. ◆

DEFAULT

```
-shared_lib_export off
```

EXAMPLE

Assume your source file contains the following code:

```
extern void not_exported();          /* not defined */
static int unexported = 1;           /* internal linkage */
int exported = 2;                    /* external linkage and defined */
void export_me() {}                  /* external linkage and defined */
```

In this case, the command line

```
MrC -shared_lib_export on myfile.c
```

generates an export file, `myfile.c.x`, that contains the following names:

```
export_me
exported
```

SEE ALSO

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

-export_list

You can use the `-export_list` option to send the export list created by the `-shared_lib_export` option to a specified file.

`-export_list` *filename*

filename Any filename.

DESCRIPTION

The `-export_list` option sends an export list to the specified file. When linking with an import library, this list would be provided to the PowerPC linker, PPCLink. For more information, see the discussions of linking in *Inside Macintosh: PowerPC System Software* and *Building and Managing Programs in MPW*, second edition.

Note
The `-export_list` option is not valid unless you also use the `-shared_lib_export on` option. ◆

DEFAULT

MrC appends a `.x` extension to the source filename (for example, `myfile.c` becomes `myfile.c.x`).

EXAMPLE

The following command creates an export list:

```
MrC -export_list mylist.x -shared_lib_export on myfile.c
```

SEE ALSO

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

Controlling Temporary Files

You can set options to control the temporary files that MrC generates. You can use the `-y` option to set the directory where you want MrC to put temporary files. The `-curdir` option sets the current working directory for the temporary files and object files (see the `-o` option, page 3-32).

-y

You can use the `-y` option to specify the directory where you want the MrC compiler to put all temporary files.

`-y` *directory*

directory	A relative or absolute pathname, with or without a terminating colon (:).

DESCRIPTION

The `-y` option specifies the directory where you want the MrC compiler to send temporary files. Any directory you specify in the pathname must already exist; MrC will not create a directory for you.

DEFAULT

MrC puts temporary files in the directory pointed to by the MPW Shell variable `"{TempFolder}"`.

EXAMPLE

The following command sends your temporary file to a directory on your RAM disk:

```
MrC -y "RAM Disk:temp:" myfile.c
```

Note
Storing temporary files in a RAM disk can decrease compilation time. ◆

-curdir

You can use the `-curdir` option to use the current (working) directory for the temporary file and object file that the MrC compiler generates.

```
-curdir
```

DESCRIPTION

The `-curdir` option specifies that MrC should put temporary files and object files in the current (working) directory.

DEFAULT

MrC writes its temporary file to `"{TempFolder}"`.

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 MrC. 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 that includes a set of headers that you want to include elsewhere in order to shorten compilation times.

A dump file created by the MrC compiler is *not* compatible with the MrCpp 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 MrC compiler.

A dump file created by the MrC compiler cannot be loaded into the MrCpp 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 MrC searches to locate header files. You can use the `-inclpath` option to control how header files are

treated. You can use the `-notOnce` option to allow MrC to read in multiple copies of the same header file.

-i

You can use the `-i` option to add a directory to the list of directories that MrC 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 MrC compiler looks for included source files. You can specify more than one `-i` option on the same command line. MrC 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 MrC to search for any included header files in the directory `::my_includes` before searching the directory `{PPCCIncludes}`:

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

SEE ALSO

See the section “Including Header Files,” beginning on page 2-8.

-inclpath

You can use the `-inclpath` option to control how included files are treated.

`-inclpath` *parameter*

The available values for *parameter* are

`normal`	Distinguish between system header files, which are given in angle brackets (<>), and include files in quotes ("").
`std`	Identical to `normal`.
`standard`	Identical to `normal`.
`ignoresys`[`tem`]	Treat system header files exactly like header files in quotes.
`nosys`[`tem`]	Identical to `ignoresys`.

DESCRIPTION

The `-inclpath` option controls how system header files and your own include files are treated. You can choose to distinguish between the two kinds of files or to treat the two the same.

DEFAULT

```
-inclpath normal
```

SEE ALSO

See the section "Handling Files," beginning on page 2-7 for more information on header files.

-notOnce

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

```
-notOnce
```

DESCRIPTION

The `-notOnce` option instructs the MrC 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 MrC to read header files only once.

SEE ALSO

See the `once` pragma, which allows you to specify directly in a header file that MrC should include it only once. This pragma is described on page 3-50.

Pragmas

Pragmas are 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. 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 680x0 alignment for data structures.

`#pragma options align=`*parameter*

The available values of *parameter* are

`power`	Use PowerPC alignment.
`mac68k`	Use 680x0 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 680x0 alignment, however, if you want data structures to be bit-for-bit compatible with 680x0 data structures created by 680x0 applications or by emulated 680x0 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 effect. MrC 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, MrC 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 680x0 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 PowerPC alignment, regardless of the type of alignment convention in effect, unless they appear inside aggregate types. In this case, MrC aligns the scalar types according to the alignment convention in effect. ◆

DEFAULT

The default is PowerPC 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 680x0 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 680x0 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 680x0 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 that 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" on page 2-9 and the description of the `-align` option on page 3-5.

noreturn

You can use the `noreturn` pragma to tell MrC 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 MrC 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 MrC always includes any file included in a precompiled header only once. You can explicitly instruct MrC to include a file more than once, using the `-notOnce` option, which is described on page 3-45.

DEFAULT

MrC 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 command-line options. You can set any number of options using the parameter values from either Table 3-1 or Table 3-2.

You can test the settings of these options using `__option(`*parameter*`)`. For example, you can test whether the `ansi` option is set using

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

Table 3-1 shows all parameter values for the `options` pragma, except for those used with warnings, which are shown in Table 3-2.

Table 3-1 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 3-45)
`chars_unsigned`	The `-char` option (page 3-7)
`pack_enums`	The `-enum` option (page 3-8) with the `min` parameter value
`struct_align`	The `-align` option (page 3-5)
`stop_at_first_err`	The `-x` option (page 3-36)
`report_all_err`	The `-x` option (page 3-36)
`generate_warn`	The `-w` option (page 3-34)
`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)

Table 3-1 Parameter values for the `options` pragma (continued)

Parameter	Related Option
`signed_pstrs`	The `-ansi` option (page 3-14)
`require_protos`	The `-proto` option (page 3-17)
`infer_protos`	The `-proto` option (page 3-17)
`mapcr`	The `-noMapCR` option (page 3-15)

Table 3-2 gives the parameter values for warnings you can use with the `options` pragma. Warnings are explained in the discussion of the `-w` option on page 3-34.

Table 3-2 Parameter values for warnings

Parameter
`warn_unintended_align`
`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 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, described on 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 and export the templates to other files.
`extern`	Do not expand the specified templates during compilation.
`static`	When expanding templates, keep them local to the current source file.

You specify one or more templates using the `template` pragma (page 3-54).

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 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. You must first specify at least one template with the `template` pragma in order to use this pragma.

DEFAULT

The default is static access.

EXAMPLE

You could set the access of the template example shown on page 3-54 to public access as follows:

```
#pragma template_access public
```

SEE ALSO

See the `template` pragma, described on page 3-54.

See the `-xa` option, described on page 3-19.

Predefined Symbols

The MrC 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 3-3 lists the ANSI predefined symbols.

Table 3-3 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).
`__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 3-4 shows the predefined symbols specific to the MrC compiler.

Table 3-4 MrC predefined symbols

Symbol	Predefined meaning
`__MRC__`	MrC is being used.
`MPW_CPLUS`	MPW C++ is being used.
`MPW_C`	MPW C is being used.
`macintosh`	Code is compiled for Macintosh.
`__MC601`	Code is compiled for the PowerPC 601 processor.
`__POWERPC`	Code is compiled for a PowerPC processor.
`__useAppleExts__`	Apple extensions are used and the `-ansi` option is not set to `strict`.
`__cplusplus`	Value is 1 if compiling in the MrCpp compiler.
	The value is not defined if compiling in the MrC compiler.
`powerc`	A compiler that generates PowerPC code.
`__powerc`	A compiler that generates PowerPC code.

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. Keep in mind that 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 Both 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 such characters 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 newline character.
- 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 ellipses (…) 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 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 or 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.**

680x0 Any member of the Motorola 68000 family of microprocessors.

680x0 alignment An alignment convention supported by MrC that matches, bit for bit, the alignment used by the MPW C and MPW C++ compilers in the 680x0 environment. See also **PowerPC alignment** and **alignment convention.**

680x0 application An application that contains code only for a 680x0 microprocessor. See also **fat application.**

680x0-based Macintosh computer Any computer containing a 680x0 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 for Information Systems—Programming Language—C* (ANSI X3.159-1991). See also **strict ANSI C.**

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.

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 for Information Systems—Programming*

Language—C++ (ANSI X3J16). This definition is based on the CFront version 3.0 from USL (UNIX System Laboratories).

error message A MrC 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.**

Extended Common Object File Format (XCOFF) The format of executable files generated by the MrC compiler.

fat application An application that contains code of two or more instruction sets. See also **680x0 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 MrC 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 MrC but that the compiler expands only if it determines a resulting speed benefit without excessive code expansion.

intrinsic ANSI C library function A library function assumed by MrC 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.

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

MrC compiler An ANSI-compliant C compiler that produces highly optimized code for the PowerPC environment.

MrCpp compiler An ANSI-compliant C++ compiler that produces highly optimized code for the PowerPC environment.

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 MrC optimization. See also **local optimization, size optimization,** and **speed optimization.**

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 MrC that aligns each element to provide the fastest memory access on the PowerPC architecture. See also **680x0 alignment** and **alignment convention.**

PowerPC 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).

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

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

PowerPC-based Macintosh computer Any computer containing a PowerPC central processing unit that runs Macintosh system software. Compare **680x0-based Macintosh computer.**

PowerPC C (PPCC) runtime library A collection of functions that support the runtime function calls generated by MrC.

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

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.

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.

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 MrC 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 MrC 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 MrC.

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 MrC 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.

XCOFF See **Extended Common Object File Format.**

Index

Symbols

Numerals

A

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

S

T

U

V

W

X

Y

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 system. 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.