MATLAB® C++ Math Library

The Language of Technical Computing

Computation

Visualization

Programming

The MATH WORKS Inc.

User's Guide

Version 2.1

How to Contact The MathWorks:

508-647-7000	Phone
508-647-7001	Fax
The MathWorks, Inc. 3 Apple Hill Drive Natick, MA 01760-2098	Mail
`http://www.mathworks.com`	Web
`ftp.mathworks.com`	Anonymous FTP server
`comp.soft-sys.matlab`	Newsgroup
`support@mathworks.com`	Technical support
`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`subscribe@mathworks.com`	Subscribing user registration
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

MATLAB C++ Math Library User's Guide

Printing History:		
	October 1995	First printing
	January 1998	Revised for Version 1.2
	January 1999	Revised for Version 2.0 (Release 11)
	September 2000	Revised for Version 2.1 (Release 12) Online only

Contents

1 Getting Started

2 Fundamentals

3 Working with MATLAB Arrays

4 Indexing into Arrays

5 Calling Library Functions

6 Using the Mathematical Operators

7 Printing, Exceptions, and Memory Management

8 Array Input and Output

9 Translating from MATLAB to C++

10 mwArray Class Interface

11 Library Routines

A Directory Organization

B Exception Classes

C Error Messages

1

Getting Started

Introduction

The MATLAB® C++ Math Library serves two separate constituencies: MATLAB programmers seeking more speed or complete independence from interpreted MATLAB, and C++ programmers who need a fast, easy-to-use matrix math library. To each, it offers distinct advantages.

MATLAB M-file programmers can write code that looks like M-file code but runs significantly faster. Because the syntax of the C++ interface is so similar to the MATLAB syntax, this performance comes at very little cost. MATLAB programmers can leverage their knowledge of M-file programming to become productive with this library very quickly. An additional advantage is that programs developed with this library do not require the interpreted MATLAB environment to execute. In addition, you may freely distribute applications you develop with the MATLAB C++ Math Library.

To C++ programmers, this library provides a natural and robust interface and a rich collection of powerful functions. MATLAB's M-file programming interface has been used by hundreds of thousands of scientists and engineers worldwide. It allows them to program the way they think, using a syntax that is simple and intuitive. Because MATLAB handles details like memory management, programmers can devote more of their mental effort to solving a problem and less to coping with the tool itself.

The ease of use that distinguishes MATLAB is a hallmark of this library as well. In addition to its natural syntax, MATLAB is easy to use because of the large number of functions it contains. Often a solution consists of little more than a single page of code. Such short programs mean easier maintenance and higher productivity.

Overview of the MATLAB C++ Math Library

The MATLAB C++ Math Library consists of approximately 400 MATLAB math functions. It includes the built-in MATLAB math functions and many of the math functions that are implemented as MATLAB M-files. The MATLAB C++ Math Library is layered on top of the MATLAB C Math Library. The major value added by this C++ layer is ease of use.

The MATLAB C++ Math Library is firmly rooted in the traditions of the MATLAB runtime environment. Programming with the MATLAB C++ Math Library is very much like writing M-files in MATLAB. While the C++ language imposes several differences, the syntax used by the MATLAB C++ Math

Library is very similar to the syntax of the MATLAB language. Like MATLAB, the MATLAB C++ Math Library provides automatic memory management, which protects the programmer from memory leaks. For a detailed comparison between MATLAB and the MATLAB C++ Math Library, see Chapter 2.

An important goal of this product is to provide a library that feels natural to both C++ programmers and MATLAB users. Achieving this goal is difficult, because there is some tension between the natural C++ programming style and the natural MATLAB style. Where it was necessary to choose between the MATLAB or C++ way of doing things, the MATLAB method usually prevailed.

The MATLAB C++ Math Library defines a set of classes and functions for the development of linear algebraic algorithms. The most important class in the MATLAB C++ Math Library is `mwArray`. This class corresponds to MATLAB's array data type. The `mwArray` class supports most MATLAB operators and all of the mathematical functions. The only operators it does not support are `\`, `./`, `.\`, `.*`, and `.^`, which are not syntactically valid in C++. These operations are accessed via function calls.

Note Do not confuse the name of the MATLAB C++ Math Library class `mwArray` with the MATLAB C Math Library data structure `mxArray`.

MATLAB is known in programming-language theory as a "functional" language: neither functions nor operators have side effects. The MATLAB C++ Math Library preserves the functional nature of MATLAB. With one exception (see "Using Indexing in Assignment Statements" in Chapter 4), expressions do not modify the arrays they contain and functions do not modify their inputs. The only way to change the value of an array is by assignment to one or more elements of the array.

The functions and operators provided by the library are vectorized. This means that they contain loops to iterate over the elements of their inputs. As a consequence, code written using this library should contain very few loops over array elements; most programs will have none.

The interface to the library is divided into three parts:

- The set of functions, or in C++ terminology the public methods, provided by the `mwArray` class
- The MATLAB mathematical functions

- A set of binary and unary mathematical operators

The bulk of the interface consists of the MATLAB math functions.

When using this library, you most often call the MATLAB mathematical functions, the operators, and the `mwArray` constructors. The public methods of `mwArray` are for the most part used internally by the library.

In general, the library code indicates that an error has occurred by raising an exception. Exception objects are subclasses of `mwException`, and thus all types of exceptions can be caught with a single `catch` statement. Each exception has an associated error message, which can be printed by placing the exception into an output stream, for example, via `cout`.

We highly recommend that you use C++ exception handling when using this library. If you do not, the first error that occurs will cause your program to terminate with a cryptic error message, such as `Unhandled exception, abnormal program termination`.

Who Should Read This Book

This book is intended to be a practical introduction to programming with the MATLAB C++ Math Library. It is written for programmers. In order to use this library, you need to understand what a function call is, how to declare a variable, what the phrase “pass by value” means, and what program control structure is. Knowledge of some common programming techniques such as reference counting helps you gain a deeper understanding of how the library works, but is not essential.

If you have never programmed before, you may find this manual difficult to read. Writing C++ programs requires a different set of skills from those required to use even a very technical program like MATLAB.

To get the most out of this document, you should be familiar with writing either C++ programs or MATLAB M-files. The annotations to the code examples assume that you know one language or the other, and often try to teach you about one language by reference to the other.

The intended audience for this manual is C++ programmers who need a matrix math library or MATLAB programmers who want the ease of M-file programming and the performance of C++. This book will not teach you how to program in either MATLAB or C++; see “Additional Sources” on page 1-7 for pointers to sources of information on these topics.

New MATLAB C++ Math Library Features

The MATLAB C++ Math Library Version 2.1 supports these new features:

- Support for the `eval` function, for expressions that do not contain variables
- Support for the `input` function, with the same restrictions as `eval`
- Performance enhancements in the core numerical routinesOver 60 new functions

Unsupported MATLAB Features

The library does not include any Handle Graphics® or Simulink® functions. For information about compiling an application that uses graphics functions, see the *MATLAB C/C++ Graphics Library User's Guide*.

In addition, the library does *not* support MATLAB objects

Changed Features

In version 2.0, any empty array could be used as an indexed deletion operator. In version 2.1, you must use the `empty()` function for indexed deletion.

MATLAB C++ Math Library Documentation

The complete documentation set for the MATLAB C++ Math Library consists of printed and online publications:

- *MATLAB C++ Math Library User's Guide*—This manual provides tutorial information about the library. This manual is also available in PDF format, accessible through the Help Desk.
- *MATLAB C++ Math Library Reference*—The reference pages for all the MATLAB C++ Math library routines are available in HTML and PDF versions, accessible through the Help Desk.

How This Book Is Organized

This chapter provides an introduction to the MATLAB C++ Math Library and tells how to install it. In addition, it includes information about building applications. The remainder of the book is organized as follows:

- Chapter 2, "Fundamentals". This chapter describes the basic concepts, assumptions, and data structures of MATLAB and the MATLAB C++ Math Library. It also provides an introduction to C++ for MATLAB users and an overview of MATLAB for C++ programmers. If you are new to MATLAB or C++, you should read this chapter.

- Chapter 3, "Working with MATLAB Arrays". Arrays are the fundamental MATLAB data type. This chapter describes how to create MATLAB arrays in your C++ program.
- Chapter 4, "Indexing into Arrays". This chapter describes how to access individual elements, or groups of elements, in an array. Using indexing you can access, modify, or delete elements in an array.
- Chapter 5, "Calling Library Functions". This chapter describes the MATLAB C++ Math Library interface to the MATLAB functions. This chapter describes how to call MATLAB functions that accept any number of input and output arguments.
- Chapter 6, "Using the Mathematical Operators". This chapter describes the difference between array and matrix operators and documents where the library overloads C++ operators and where you must call functions that are equivalent to a MATLAB operator.
- Chapter 7, "Printing, Exceptions, and Memory Management". This chapter describes how to use the MATLAB C++ Math Library routines in a C++ program. The chapter includes specific information about handling errors and writing your own print handler.
- Chapter 8, "Array Input and Output". This chapter describes the library's three input/output mechanisms: input and output streams, `fprintf()` and `fscanf()`, and `load()` and `save()`.
- Chapter 9, "Translating from MATLAB to C++". This chapter compares the MATLAB language to C++.
- Chapter 10, "mwArray Class Interface". This chapter documents the public interface of the `mwArray` class.
- Chapter 11, "Library Routines". This chapter groups the more than 400 library functions into functional categories and provides a short description of each function.
- Appendix A: Directory Organization. Installing the MATLAB C++ Math Library creates several new directories on your computer. This appendix provides a road map to the directories and their contents for PC systems running Microsoft Windows and UNIX workstations.
- Appendix B: Exception Classes. This appendix describes the hierarchy of exception classes defined by the library.

- Appendix C: Error Messages. This appendix provides a reference to the error messages issued by the library.

Accessing Online Reference Documentation

To access the MATLAB online documentation, select the Help option from the MATLAB menu bar. MATLAB C Math Library documentation is available in in HTML and PDF formats.

To look up the syntax and behavior for a C Math Library function, refer to the *MATLAB C Math Library Reference*. This reference gives you access to a reference page for each function. Each page presents the function's C syntax and links you to the online *MATLAB Function Reference* page for the corresponding MATLAB function.

If you are a stand-alone Math Library user:

1 Open the HTML file `<matlab>/help/mathlib.html` with your Web browser, where `<matlab>` is the top-level directory where you installed the MATLAB C++ Math Library.

2 Select *MATLAB C++ Math Library Reference*.

Additional Sources

- Release notes for the MATLAB C++ Math Library `<matlab>/extern/examples/cppmath`
- *MATLAB C Math Library User's Guide*
- *MATLAB C Math Library Reference*
- *MATLAB Application Program Interface Guide*
- *MATLAB Application Program Interface Reference*
- *MATLAB Function Reference*
- *Installation Guide for UNIX*
- *Installation Guide for PC*

For general information about C++ programming language, see:

C++ Primer 2nd Ed., Lippman, Stanley, Addison Wesley, 1993

Getting Started Quickly

Depending on your experience with other MathWorks products, your knowledge of C++, and your goals, you may not need to read this book in its entirety. If you are eager to get started, the following sections give you a solid understanding of programming with the MATLAB C++ Math Library.

- “Example Program: Handling Exceptions (ex5.cpp)” in Chapter 7 demonstrates most of the features of the library: creating matrices, writing and calling your own functions, printing matrices, and handling errors.
- “Building C++ Applications” on page 1-13 explains how to build and run the example programs with the `mbuild` script. This is highly recommended reading.
- “Differences Between C++ and MATLAB” in Chapter 9 details the differences between MATLAB and C++, and provides information about the MATLAB C++ Math Library.
- ““Working with MATLAB Arrays” in Chapter 3,” explains how to create an array and why some ways are more efficient than others. MATLAB C++ Math Library arrays are considerably different from C++ two-dimensional arrays. Study this section with care.
- ““Indexing into Arrays” in Chapter 4,” explains how to apply subscripts to arrays. The indexing facility, which is a fundamental part of the MATLAB C++ Math Library, is quite powerful, but may occasionally give you unexpected results if you do not understand how and why it works the way it does.

Reading only these sections means you omit a lot of detail and risk stumbling through parts of the library that you don’t understand. However, if you read and understand these six sections, you can do useful work with this library.

Installing the C++ Math Library

The MATLAB C++ Math Library is available on UNIX workstations and PCs running Microsoft Windows (Windows 95/98 and Windows NT). The installation process is different for each platform.

The MATLAB C++ Math Library contains the MATLAB C Math Library. If you already have the MATLAB C Math Library, the installation program will overwrite your existing copy of the MATLAB C Math Library with new libraries and header files. You'll still be able to use both the MATLAB C++ Math Library and the MATLAB C Math Library.

Note that the MATLAB C++ Math Library (and the MATLAB C Math Library, for that matter) runs on only those platforms (processor and operating system combinations) on which MATLAB runs. In particular, the Math Libraries do not run on DSP or other embedded systems boards, even if those boards are controlled by a processor that is part of a system on which MATLAB runs.

Installation with MATLAB

If you are a licensed user of MATLAB, there are no special requirements for installing the MATLAB C++ Math Library. Follow the instructions in the MATLAB *Installation Guide* for your specific platform:

- *Installation Guide for UNIX*
- *Installation Guide for PC*

The C/C++ Math Library will appear as one of the installation choices that you can select as you proceed through the installation screens.

Before you begin installing the MATLAB C/C++ Math Library, you must obtain from The MathWorks a Personal License Password (PLP) and, if you are installing the library in a concurrent access environment, a valid License File. These are usually supplied by fax or e-mail. If you have not already received a License File or PLP, contact The MathWorks via:

- The Web at `www.mathworks.com`. On the MathWorks site, click on the MATLAB Access option, log in to the Access home page, and follow the instructions. MATLAB Access membership is free of charge and available to all customers.
- E-mail at `service@mathworks.com`

- Telephone at 508-647-7000; ask for Customer Service
- Fax at 508-647-7001

MATLAB Access members can obtain the necessary license data via the Web (`www.mathworks.com`). Click on the MATLAB Access icon and log in to the Access home page. MATLAB Access membership is free of charge.

Installation Without MATLAB

The installation process for installing the C++ Math Library in stand-alone mode is identical to the process for installing MATLAB and its toolboxes. Although you are not actually installing MATLAB, you can still follow the instructions in the MATLAB *Installation Guide* for your specific platform.

Verifying a UNIX Installation

To verify that the MATLAB C++ Math Library has been installed correctly, use the `mbuild` script, which is documented in "Building a Stand-Alone Application on UNIX" on page 1-15, to verify that you can build one of the example applications. Be sure to use `mbuild` before calling Technical Support.

To spot check that the installation worked, `cd` to the directory `<matlab>/extern/include/cpp`, where `<matlab>` symbolizes the MATLAB root directory. Look for the file `matlab.hpp`.

Verifying a PC Installation

When installing a C++ compiler to use in conjunction with the Math Library, install both the DOS and Windows targets and the command line tools.

The C++ Math Library installation adds

```
<matlab>\bin
```

to your `$PATH` environment variable, where `<matlab>` symbolizes the MATLAB root directory. The `BIN` directory contains the DLLs required by stand-alone applications. After installation, reboot your machine if necessary.

To verify that the MATLAB C++ Math Library has been installed correctly, use the `mbuild` script, which is documented in "Building a Stand-Alone Application on PCs" on page 1-23, to verify that you can build one of the example applications. Be sure to use `mbuild` before calling Technical Support.

Installing Your C++ Compiler

To use the MATLAB C++ Math Library, you need to have a C++ compiler installed on your system. If you are having trouble installing your C++ compiler or getting it to work properly, please contact the manufacturer of that compiler.

The technical support number for each compiler vendor is listed in the documentation for each compiler. Many compiler vendors also have home pages on the World-Wide Web; in particular, Borland, Microsoft, and Watcom. Contact them at `www.borland.com`, `www.microsoft.com`, and `www.watcom.com` respectively.

Note The MATLAB C++ Math Library makes use of both templates and exceptions. Make sure that your compiler supports these C++ language features. If it does not support templates, you can't use the MATLAB C++ Math Library.

Compiler Configuration on Microsoft Windows

This table provides information regarding the installation and configuration of a C++ compiler on your system.

Description	Comment
Installation options	We recommend that you do a full installation of your compiler. If you do a partial installation, you might omit a component that the MATLAB C++ Math Library relies on.
Installing DGB files	For the purposes of the MATLAB C++ Math Library, it is not necessary to install DBG (debugger) files. However, you may need them for other purposes.
MFC	MFC (Microsoft Foundation Classes) are not required.
16-bit DLL/executables	Not required.

Description	Comment
ActiveX	Not required.
Running from the command line	Make sure you select all relevant options for running your compiler from the command line.
Updating the registry	If your installer gives you the option of updating the registry, you should let it do so.
Recording the root directory of your C/C++ compiler	Record the complete path to where your C/C++ compiler has been installed, for example, `C:\devstudio`.

Building C++ Applications

This section:

- Provides an overview of building C++ applications using the MATLAB `mbuild` utility.
- Explains how to build stand-alone C++ applications on UNIX systems
- Explains how to build stand-alone C++ applications on systems running Microsoft Windows

For information about packaging a stand-alone application for redistribution, see "Distributing Stand-Alone Applications" on page 1-36. For information about building applications without using `mbuild`, see "Linking Applications Without mbuild" on page 1-35.

Note You may freely distribute applications you develop with the MATLAB C++ Math Library.

Overview

To build a stand-alone application using the MATLAB C++ Math Library, you must supply your C++ compiler with the correct set of compiler and linker options (or switches). To help you, The MathWorks provides a command line utility called `mbuild`. The `mbuild` script makes it easy to:

- Set your compiler and linker settings
- Change compilers or compiler settings
- Switch between C and C++ development
- Build your application

On UNIX and Microsoft Windows systems, follow these steps to build C++ applications with `mbuild`:

1. Verify that `mbuild` can create stand-alone applications.
2. Build your application.

You only need to reconfigure if you change compilers or upgrade your current compiler.

Compiler Options Files

`mbuild` stores compiler and linker settings in an options file. Options files contain the required compiler and linker settings for your particular C++ compiler. The MathWorks provides options files for every supported C++ compiler.

Much of the information on options files in this chapter is provided for those users who may need to modify an options file to suit their specific needs. Many users never have to be concerned with how the options files work.

Building a Stand-Alone Application on UNIX

This section:

- Explains how to set-up your build environment
- Describes how to compile and link C++ source code into a stand-alone UNIX application.

This section also contains information about packaging your application for distribution.

Configuring the Build Environment

`mbuild` determines whether to compile in C or C++ by examining the type of files you are compiling. Table 1-1 shows the supported file extensions. If you include both C and C++ files, `mbuild` uses the C++ compiler and the MATLAB C++ Math Library. If `mbuild` cannot deduce from the file extensions whether to compile in C or C++, `mbuild` invokes the C compiler.

Table 1-1: UNIX File Extensions for mbuild

Language	Extension(s)
C	`.c`

C++	`.cpp`
	`.C`
	`.cxx`
	`.cc`

Note You can override the language choice that is determined from the extension by using the `-lang` option of `mbuild`. For more information about this option, as well as all of the other `mbuild` options, see Table 1-2.

Locating Options Files

`mbuild` locates your options file by searching the following:

- The current directory

- `$HOME/.matlab/R12`
- `<matlab>/bin`

`mbuild` uses the first occurrence of the options file it finds. If no options file is found, `mbuild` displays an error message.

Using the System Compiler

If your supported C++ compiler is installed on your system, you are ready to create C++ stand-alone applications. To create a stand-alone C++ application, you can simply enter

```
mbuild filename.cpp
```

This simple method works for the majority of users. Assuming `filename.cpp` contains a `main` function, this example uses the system's compiler as your default compiler for creating your stand-alone application.

- If you are a user who does not need to change C or C++ compilers, or you do not need to modify your compiler options files, you can skip ahead in this section to "Building an Application."
- If you need to know how to select a different compiler or change the options file, continue with this section.

Changing the Default Compiler

You need to use the `setup` option if you want to change your default compiler. At the UNIX prompt type:

```
mbuild -setup
```

The `setup` option creates a user-specific options file for your ANSI C or C++ compiler. Using the `setup` option sets your default compiler so that the new compiler is used every time you use the `mbuild` script.

Note The options file is stored in the MATLAB subdirectory of your home directory, for example, `$HOME/.matlab/R12/mbuildopts.sh`. This allows each user to have a separate `mbuild` configuration.

Executing `mbuild -setup` presents a list of options files currently included in the `bin` subdirectory of MATLAB.

```
mbuild -setup

Using the 'mbuild -setup' command selects an options file that is
placed in ~/.matlab/R12 and used by default for 'mbuild'. An
options file in the current working directory or specified on the
command line overrides the default options file in ~/.matlab/R12.

Options files control which compiler to use, the compiler and link
command options, and the runtime libraries to link against.

To override the default options file, use the 'mbuild -f' command
(see 'mbuild -help' for more information).

The options files available for mbuild are:

  1: /matlab/bin/mbuildopts.sh :
         Build and link with MATLAB C/C++ Math Library
```

If there is more than one options file, you can select the one you want by entering its number and pressing **Return**. If there is only one options file available, it is automatically copied to your MATLAB directory if you do not already have an `mbuild` options file. If you already have an `mbuild` options file, you are prompted to overwrite the existing one.

Modifying the Options File

Another use of the `setup` option is if you want to change your options file settings. For example, if you want to make a change to the current linker settings, or you want to disable a particular set of warnings, you should use the `setup` option.

If you need to change the options that `mbuild` passes to your compiler or linker, you must first run

```
mbuild -setup
```

which copies a master options file to your local MATLAB directory, typically `$HOME/.matlab/R12/mbuildopts.sh`.

If you need to see which options `mbuild` passes to your compiler and linker, use the verbose option, `-v`, as in

```
mbuild -v filename1 [filename2 …]
```

to generate a list of all the current compiler settings.

To change the options, use an editor to make changes to your options file, which is in your local MATLAB directory. Your local MATLAB directory is a user-specific, MATLAB directory in your individual home directory that is used specifically for your individual options files.

You can also embed the settings obtained from the verbose option of `mbuild` into an integrated development environment (IDE) or makefile that you need to maintain outside of MATLAB. Often, however, it is easier to call `mbuild` from your makefile. See your system documentation for information on writing makefiles.

Note Any changes made to the local options file will be overwritten if you execute `mbuild -setup` again. To make the changes persist through repeated uses of `mbuild -setup`, you must edit the master file itself, `<matlab>/bin/mbuildopts.sh`.

Temporarily Changing the Compiler

To temporarily change your C or C++ compiler, use the `-f` option, as in

```
mbuild -f <options_file> filename.cpp [filename]
```

The `-f` option tells the `mbuild` script to use the options file, `<file>`. If `<file>` is not in the current directory, then `<file>` must be the full pathname to the desired options file. Using the `-f` option tells the `mbuild` script to use the specified options file for the current execution of `mbuild` only; it does not reset the default compiler.

Building an Application

There is C++ source code for example `ex1.cpp` included in the `<matlab>/extern/examples/cppmmath` directory, where `<matlab>` represents the top-level directory where MATLAB is installed on your system. To verify that `mbuild` is properly configured on your system to create stand-alone applications, copy `ex1.cpp` to your local directory and `cd` to that directory. Then, at the UNIX prompt, enter:

```
mbuild ex1.cpp
```

This should create the file called ex1. Stand-alone applications created on UNIX systems do not have any extensions. If you have a problem using mbuild, see "Troubleshooting mbuild" on page 1-33.

Locating Shared Libraries

Before you can run your stand-alone application, you must tell the system where the API and C++ shared libraries reside. This table provides the necessary UNIX commands depending on your system's architecture.

Architecture	Command
HP700	`setenv SHLIB_PATH <matlab>/extern/lib/hp700:$SHLIB_PATH`
IBM RS/6000	`setenv LIBPATH <matlab>/extern/lib/ibm_rs:$LIBPATH`
All others	`setenv LD_LIBRARY_PATH <matlab>/extern/lib/<arch>:$LD_LIBRARY_PATH`
	where: <matlab> is the MATLAB root directory <arch> is your architecture (i.e., alpha, glnx86, sgi, sol2)

It is convenient to place this command in a startup script such as ~/.cshrc. Then, the system will be able to locate these shared libraries automatically, and you will not have to re-issue the command at the start of each login session. The best choice is to place the libraries in ~/.login, which only gets executed once.

Note On all UNIX platforms, the C/C++ libraries are shipped as shared object (.so) files or shared libraries (.sl). Any stand-alone application must be able to locate the C/C++ libraries along the library path environment variable (SHLIB_PATH, LIBPATH, or LD_LIBRARY_PATH) in order to be loaded. Consequently, to share a stand-alone application with another user, you must provide all of the required shared libraries. For more information about the required shared libraries for UNIX, see "Building a Stand-Alone Application on PCs" on page 1-23.

Running Your Application

To launch your application, enter its name on the command line. For example,

```
ex1

  [
        1        3        5 ;
        2        4        6
  ]

  [
        1        4 ;
        2        5 ;
        3        6
  ]

Please enter a matrix:
```

mbuild Options

The `mbuild` script supports various options that allow you to customize the building and linking of your code. Many users do not need to know additional details about the `mbuild` script; they use it in its simplest form. The following information is provided for those users who require more flexibility with the tool.

The `mbuild` syntax and options are

```
mbuild [-options] filename1 [filename2 …]
```

Table 1-2: mbuild Options on UNIX

Option	Description
`-c`	Compile only; do not link.
`-D<name>[=<def>]`	Define C++ preprocessor macro `<name>` [as having value `<def>`].
`-f <optionsfile>`	Use `<file>` to override the default options file; `<file>` is a full pathname if it is not in the current directory.

Table 1-2: mbuild Options on UNIX (Continued)

Option	Description
`-g`	Build an executable with debugging symbols included.
`-h[elp]`	Help; prints a description of `mbuild` and the list of options.
`-I<pathname>`	Include `<pathname>` in the list of directories to search for header files.
`-inline`	Inlines matrix accessor functions (mx*). The generated MEX function may not be compatible with future versions of MATLAB.
`-l<file>`	Link against library `lib<file>`.
`-L<pathname>`	Include `<pathname>` in the list of directories to search for libraries.
`-lang <language>`	Override language choice implied by file extension. `<language>` =`c` for C `cpp` for C++ This option is necessary when you use an unsupported file extension, or when you pass all `.o` files and libraries.
`<name>=<def>`	Override options file setting for variable `<name>`. If `<def>` contains spaces, enclose it in single quotes, for example, `CFLAGS='opt1 opt2'`. The definition, `<def>`, can reference other variables defined in the options file. To reference a variable in the options file, prepend the variable name with a `$`, for example, `CFLAGS='$CFLAGS opt2'`.
`-n`	No execute flag. Using this option displays the commands that compile and link the target but does not execute them.

Table 1-2: mbuild Options on UNIX (Continued)

Option	Description
`-outdir <dirname>`	Place any generated object, resource, or executable files in the directory `<dirname>`. Do not combine this option with `-output` if the `-output` option gives a full pathname.
`-output <name>`	Create an executable named `<name>`. (An appropriate executable extension is automatically appended.)
`-O`	Build an optimized executable.
`-setup`	Set up the default compiler and libraries. This option should be the only argument passed.
`-U<name>`	Undefine C++ preprocessor macro `<name>`.
`-v`	Verbose; print all compiler and linker settings.

Building a Stand-Alone Application on PCs

This section:

- Explains how to set-up your build environment
- Describes how to compile and link C++ source code into a stand-alone UNIX application.

This section also contains information about packaging your application for distribution.

Configuring the Build Environment

`mbuild` determines whether to compile in C or C++ by examining the type of files you are compiling. Table 1-3, Windows File Extensions for mbuild shows the file extensions that `mbuild` interprets as indicating C or C++ files. If you include both C and C++ files, `mbuild` uses the C++ compiler and the MATLAB C++ Math Library. If `mbuild` cannot deduce from the file extensions whether to compile in C or C++, `mbuild` invokes the C compiler.

Table 1-3: Windows File Extensions for mbuild

Language	Extension(s)
C	`.c`

C++	`.cpp`
	`.cxx`
	`.cc`

Note You can override the language choice that is determined from the extension by using the `-lang` option of `mbuild`. For more information about this option, as well as all of the other `mbuild` options, see Table 1-5.

Locating Options Files

To locate your options file (`compopts.bat`), the `mbuild` script searches the following:

- The current directory
- The user Profiles directory
- <matlab>\bin

mbuild uses the first occurrence of the options file it finds. If no options file is found, mbuild searches your machine for a supported C++ compiler and uses the factory default options file for that compiler. If multiple compilers are found, you are prompted to select one.

The User Profile Directory Under Windows. The Windows user Profiles directory is a directory that contains user-specific information such as Desktop appearance, recently used files, and Start Menu items. The mbuild utility stores the options file (compopts.bat) it create during the -setup process in a subdirectory of your user Profiles directory, named Application Data\MathWorks\MATLAB\R12.

Under Windows NT and Windows 95/98/2000 with user profiles enabled, your user profile directory is %windir%\Profiles\username. Under Windows 95/98 with user profiles disabled, your user profile directory is %windir%. Under Windows 95/98, you can determine whether or not user profiles are enabled by using the Passwords control panel.

Systems with Exactly One C/C++ Compiler

If your supported C++ compiler is installed on your system, you are ready to create C++ stand-alone applications. On systems where there is exactly one C++ compiler available to you, the mbuild utility automatically configures itself for the appropriate compiler. So, for many users, to create a C++ stand-alone application, you can simply enter

```
mbuild filename.cpp
```

This simple method works for the majority of users. It uses your installed C++ compiler as your default compiler for creating your stand-alone applications:

- If you are a user who does not need to change compilers, or you do not need to modify your compiler options files, you can skip ahead in this section to “Building an Application.”
- If you need to know how to change the options file or select a different compiler, continue with this section.

Systems with More than One Compiler

On systems where there is more than one C++ compiler, the `mbuild` utility lets you select which of the compilers you want to use. Once you choose your C++ compiler, that compiler becomes your default compiler and you no longer have to select one when you compile your stand-alone applications.

For example, if your system has both the Borland and Watcom compilers, when you enter for the first time

```
mbuild filename.cpp
```

you are asked to select which compiler to use.

```
mbuild has detected the following compilers on your machine:

[1] : Borland compiler in T:\Borland\BC.500
[2] : MSVC compiler in T:\DevStudio\c.106

[0] : None

Please select a compiler. This compiler will become the default:
```

Select the desired compiler by entering its number and pressing **Return**. You are then asked to verify your information.

Changing the Default Compiler

To change your default C++ compiler, you select a different options file. You can do this at any time by using the `setup` command.

This example shows the process of changing your default compiler to the Microsoft Visual C/C++ Version 6.0 compiler.

```
mbuild -setup

Please choose your compiler for building standalone MATLAB
applications.

Would you like mbuild to locate installed compilers [y]/n? n

Choose your C/C++ compiler:
[1] Borland C/C++          (version 5.0, 5.2, or 5.3)
[2] Microsoft Visual C/C++ (version 4.2, 5.2, or 6.0)
```

```
[0] None

Compiler: 2

Choose the version of your C/C++ compiler:
[1] Microsoft Visual C/C++ 4.2
[2] Microsoft Visual C/C++ 5.0
[3] Microsoft Visual C/C++ 6.0

version: 3

Your machine has a Microsoft Visual C/C++ compiler located at
D:\Program Files\DevStudio6.
Do you want to use this compiler [y]/n? y

Please verify your choices:

Compiler: Microsoft Visual C/C++ 6.0
Location: D:\Program Files\DevStudio6

Are these correct?([y]/n): y

The default options file:
"C:\WINNT\Profiles\username
\Application Data\MathWorks\MATLAB\R12\compopts.bat" is being
updated...
```

If the specified compiler cannot be located, you are given the message:

```
The default location for compiler-name is directory-name,
but that directory does not exist on this machine.
Use directory-name anyway [y]/n?
```

Using the `setup` option sets your default compiler so that the new compiler is used every time you use the `mbuild` script.

Modifying the Options File

Another use of the `setup` option is if you want to change your options file settings. For example, if you want to make a change the current linker settings, or you want to disable a particular set of warnings, use the `setup` option.

The `setup` option copies the appropriate options file to your user profile directory and names it `compopts.bat`. Make your user-specific changes to `compopts.bat` in the user profile directory and save the modified file. This sets your default compiler's options file to your specific version.

Table 1-4, Compiler Options Files on the PC lists the names of the PC master options files included in this release of the MATLAB C++ Math Library.

If you need to see which options `mbuild` passes to your compiler and linker, use the verbose option, `-v`, as in

```
mbuild -v filename1 [filename2 …]
```

to generate a list of all the current compiler settings used by `mbuild`.

You can also embed the settings obtained from the verbose option into an integrated development environment (IDE) or makefile that you need to maintain outside of MATLAB. Often, however, it is easier to call `mbuild` from your makefile. See your system documentation for information on writing makefiles.

Note Any changes that you make to the local options file `compopts.bat` will be overwritten the next time you run `mbuild -setup`. If you want to make your edits persist through repeated uses of `mbuild -setup`, you must edit the master file itself. The master options files are located in `<matlab>\bin\win32\mbuild`..

Table 1-4: Compiler Options Files on the PC

Compiler	Master Options File
Borland C/C++, Version 5.0	`bcccompp.bat`
Borland C/C++, Version 5.2	`bcc52compp.bat`
Borland C++Builder 3.0	`bcc53compp.bat`
Borland C++Builder 4.0	`bcc54compp.bat`
Borland C++Builder 5.0	`bcc55compp.bat`

Table 1-4: Compiler Options Files on the PC

Compiler	Master Options File
Lcc 2.4 (bundled with MATLAB)	`lcccompp.bat`
Microsoft Visual C/C++, Version 5.0	`msvc50compp.bat`
Microsoft Visual C/C++, Version 6.0	`msvc60compp.bat`

Combining Customized C and C++ Options Files

The options files for `mbuild` have changed as of MATLAB 5.3 (Release 11) so that the same options file can be used to create both C and C++ stand-alone applications. If you have modified your own separate options files to create C and C++ applications, you can combine them into one options file.

To combine your existing options files into one universal C and C++ options file:

1. Copy from the C++ options file to the C options file all lines that set the variables `COMPFLAGS`, `OPTIMFLAGS`, `DEBUGFLAGS`, and `LINKFLAGS`.
2. In the C options file, within just those copied lines from step 1, replace all occurrences of `COMPFLAGS` with `CPPCOMPFLAGS`, `OPTIMFLAGS` with `CPPOPTIMFLAGS`, `DEBUGFLAGS` with `CPPDEBUGFLAGS`, and `LINKFLAGS` with `CPPLINKFLAGS`.

This process modifies your C options file to be a universal C/C++ options file.

Temporarily Changing the Compiler

To temporarily change your C++ compiler, use the `-f` option, as in

```
mbuild -f <file> …
```

The `-f` option tells the `mbuild` script to use the options file, `<file>`. If `<file>` is not in the current directory, then `<file>` must be the full pathname to the desired options file. Using the `-f` option tells the `mbuild` script to use the specified options file for the current execution of `mbuild` only; it does not reset the default compiler.

Building an Application

C++ source code for example `ex1.cpp` is included in the `<matlab>\extern\examples\cppmath` directory; `<matlab>` represents the top-level directory where MATLAB is installed on your system. To verify that `mbuild` is properly configured on your system to create stand-alone applications, enter at the DOS prompt:

```
mbuild ex1.cpp
```

This should create the file called `ex1.exe`. Stand-alone applications created on Windows 95 or Windows NT always have the extension `.exe`. The created application is a 32-bit Microsoft Windows console application. If you have a problem using mbuild, see “Troubleshooting mbuild” on page 1-33.

Shared Libraries (DLLs)

All the WIN32 Dynamic Link Libraries (DLLs) for the MATLAB C++ Math Library are in the directory

```
<matlab>\bin
```

The `.def` files for the Microsoft and Borland compilers are in the `<matlab>\extern\include` directory. Import libraries for supported compilers can be found in `<matlab>\extern\lib\win32\<compiler>`.

Before running a stand-alone application, you must ensure that the directory containing the DLLs is on your path. The directory must be on your operating system `$PATH` environment variable. On Windows 95, set the value in your `autoexec.bat` file; on Windows NT, use the Control Panel to set it.

Running Your Application

You can now run your stand-alone application by launching it from the command line. For example,

```
ex1

  [
         1        3        5 ;
         2        4        6
  ]

  [
```

```
         1        4 ;
         2        5 ;
         3        6
   ]

Please enter a matrix:
```

mbuild Options

The mbuild script supports various options that allow you to customize the building and linking of your code. Many users do not need to know any additional details of the mbuild script; they use it in its simplest form. The following information is provided for those users who require more flexibility with the tool.

The mbuild syntax and options are

```
mbuild [-options] filename1 [filename2 …]
```

Table 1-5: mbuild Options on Microsoft Windows

Option	Description
@filename	Replace @filename on the mbuild command line with the contents of filename. filename is a response file, i.e., a text file that contains additional command line options to be processed.
-c	Compile only; do not link.
-D<name> [#<def>]	Define C preprocessor macro <name> as having value [#<def>]
-f <file>	Use <file> as the options file; <file> is a full pathname if it is not in the current directory.
-g	Build an executable with debugging symbols included.

Table 1-5: mbuild Options on Microsoft Windows (Continued)

Option	Description
-h[elp]	Help; prints a description of mbuild and the list of options.
-I<pathname>	Include <pathname> in the list of directories to search for header files.
<name>#<def>	Override options file setting for variable <name>. This option is equivalent to <ENV_VAR>#<val>, which temporarily sets the environment variable <ENV_VAR> to <val> for the duration of the call to mex. <val> can refer to another environment variable by prepending the name of the variable with a $, e.g., COMPFLAGS#"$COMPFLAGS -myswitch".
-inline	Inlines matrix accessor functions (mx*). The generated MEX-function may not be compatible with future versions of MATLAB.
-lang <language>	Override language choice implied by file extension. <language> =c for C cpp for C++ This option is necessary when you use an unsupported file extension, or when you pass all .o files and libraries.
-n	No execute flag. Using this option displays the commands that compile and link the target but does not execute them.

Table 1-5: mbuild Options on Microsoft Windows (Continued)

Option	Description
`-outdir <dirname>`	Place any generated object, resource, or executable files in the directory `<dirname>`. Do not combine this option with `-output` if the `-output` option gives a full pathname.
`-output <name>`	Create an executable named `<name>`. (An appropriate executable extension is automatically appended.)
`-O`	Build an optimized executable.
`-setup`	Set up the default compiler and libraries. This option should be the only argument passed.
`-U<name>`	Undefine C++ preprocessor macro `<name>`.
`-v`	Verbose; print all compiler and linker settings.

Troubleshooting mbuild

This section identifies some of the more common problems that may occur when configuring `mbuild` to create applications.

Options File Not Writable

When you run `mbuild -setup`, `mbuild` makes a copy of the appropriate options file and writes some information to it. If the options file is not writable, you are asked if you want to overwrite the existing options file. If you choose to do so, the existing options file is copied to a new location and a new options file is created.

Directory or File Not Writable

If a destination directory or file is not writable, ensure that the permissions are properly set. In certain cases, make sure that the file is not in use.

mbuild Generates Errors

On UNIX, if you run `mbuild` *`filename`* and get errors, it may be because you are not using the proper options file. Run `mbuild -setup` to ensure proper compiler and linker settings.

Compiler and/or Linker Not Found

On PCs running Windows, if you get errors such as `Bad command or filename` or `File not found`, make sure the command line tools are installed and the path and other environment variables are set correctly.

mbuild Not a Recognized Command

If `mbuild` is not recognized, verify that `<matlab>\bin` is on your path. On UNIX, it may be necessary to rehash.

Cannot Locate Your Compiler (PC)

If `mbuild` has difficulty locating your installed compilers, it is useful to know how it goes about finding compilers. `mbuild` automatically detects your installed compilers by first searching for locations specified in the following environment variables:

- `BORLAND` for the Borland C/C++ Compiler, Version 5.0 and 5.2, and Borland C++Builder, Version 3.0, 4.0, and 5.0.

- MSVCDIR for Microsoft Visual C/C++, Version 5.0 or 6.0
- MSDEVDIR for Microsoft Visual C/C++, Version 4.2

Next, mbuild searches the Windows Registry for compiler entries. Note that Watcom does not add an entry to the registry.

Internal Error When Using mbuild -setup (PC)

Some antivirus software packages such as Cheyenne AntiVirus and Dr. Solomon may conflict with the mbuild -setup process. If you get an error message during mbuild -setup of the following form

```
mex.bat: internal error in sub get_compiler_info(): don't
recognize <string>
```

then you need to disable your antivirus software temporarily and rerun mbuild -setup. After you have successfully run the setup option, you can re-enable your antivirus software.

Verification of mbuild Fails

If none of the previous solutions addresses your difficulty with mbuild, contact Technical Support at The MathWorks at support@mathworks.com or 508-647-7000.

Linking Applications Without mbuild

To build the examples or your own applications without mbuild, compile the file with a robust C++ compiler. The compiler you use must support both templates and exceptions. Set the include file search path to contain the directory that contains the file matlab.hpp; compilers typically use the -I switch to add directories to the include file search path. See Appendix A to determine where matlab.hpp is installed. Link the resulting object files against the libraries in this order:

1 MATLAB C++ Math Library (libmatpp on UNIX; libmatp* on Windows, where * is replaced by the suffix for your compiler)

2 MATLAB M-File Math Library (libmmfile)

3 MATLAB Built-In Library (libmatlb)

4 MATLAB MAT-file Library (libmat)

5 MATLAB Array Access and Creation Library (libmx)

6 Standard C Math Library (libm)

Specifying the libraries in the wrong order on the command line typically causes linker errors.

On some platforms, additional libraries are necessary; see the platform-specific section of the mbuild script for the names and order of these libraries on the platforms we support.

On PCs, import libraries can be found in <matlab>\extern\lib\win32\<compiler>.

Distributing Stand-Alone Applications

You may freely distribute applications you develop with the MATLAB C++ Math Library, subject to The MathWorks software license agreement. When you package your application for distribution, remember to include, along with your application executable, these additional files:

- The contents, if any, of a directory named `bin`, created by `mbuild` in the same directory as your application executable
- Any custom MEX files your application uses
- All the MATLAB math run-time libraries

To make packaging an application easier, the C++ math library has prepackaged all the necessary MATLAB run-time libraries into a single, self-extracting archive file. For more information about how you can use this archive, see “Packaging the MATLAB Math Run-Time Libraries”. For information about how customers who receive your application can use this archive, see “Installing Your Application” on page 1-37.

Packaging the MATLAB Math Run-Time Libraries

The MATLAB C++ Math library has prepackaged all the MATLAB run-time libraries required by stand-alone applications into a single, self-extracting archive file, called the MATLAB Math and Graphics Run-Time Library Installer. Instead of including all the run-time libraries individually in your stand-alone application distribution package, you can simply include this archive file.

The following table lists the name of the archive file for both PCs and UNIX systems. In the table `$MATLAB` represents your MATLAB installation directory and `$ARCH` represents your UNIX platform.

Platform	MATLAB Math and Graphics Run-Time Library Installer
UNIX systems	`$MATLAB/extern/lib/$ARCH/mglinstaller`
PCs	`$MATLAB\extern\lib\win32\mglinstaller.exe`

Installing Your Application

To install your application, your customers must:

- Run the MATLAB Math and Graphics Run-Time Library Installer. This program extracts the libraries from the archive and installs them in subdirectories of a directory specified by the user.
- Add the `bin/$ARCH` subdirectory to their path. This is the only MATLAB Math and Graphics Run-time library subdirectory that needs to be added to the path.

Note If a customer already has the MATLAB math and graphics run-time libraries installed on their system, they do not need to reinstall them. They only need to ensure that the library search path is configured correctly.

On UNIX Systems

On UNIX systems, your customer runs the MATLAB Math and Graphics Run-Time Library Installer by executing the `mglinstaller` command at the system prompt. Your customer can specify the name of the directory into which they want to install the libraries. By default, the installer puts the files in the current directory.

After the installer unpacks and uncompresses the libraries, your customers must add the name of the `bin/$ARCH` subdirectory to the `LD_LIBRARY_PATH` environment variable. (The equivalent variable on HP-UX systems is the `SHLIB_PATH` and `LIBPATH` on IBM AIX systems.)

For example, if a customer working on a Linux system specifies the installation directory `mgl_runtime_dir`, then they must add `mgl_runtime_dir/bin/glnx86` to the `LD_LIBRARY_PATH` environment variable.

On PCs

On PCs, your customer can run the MATLAB Math and Graphics Run-Time Library Installer by double-clicking on the `mglinstaller.exe` file. Your customer can specify the name of the directory into which they want to install the libraries. By default, the installer puts the files in the current directory.

After the installer unpacks and uncompresses the libraries, your customers must add the `bin\win32` subdirectory to the system path variable (`PATH`).

For example, if your customer specifies the installation directory mgl_runtime_dir, then they must add mgl_runtime_dir\bin\win32 to PATH.

Problem Starting Stand-Alone Application

Your application may compile successfully but fail when you or one of your customers tries to start it. If you run the application from a DOS command window, you or one of your customers may see an error message such as:

```
The ordinal #### could not be located in the dynamic-link library
dforrt.dll.
```

To fix this problem, locate the files named dforrt.dll or dformd.dll in your Windows system directory and replace them with the versions of these files in the <MATLAB>\bin\win32 directory, where <MATLAB> represents the name of your MATLAB installation directory.

This same solution works for customers of your application who encounter the same problem; however, they can replace the versions of these files in the Windows system directory with the versions they find in <MGLRUNTIMELIBRARY>\bin\win32 directory, where <MGLRUNTIMELIBRARY> is the name of the directory in which they installed the MATLAB Math and Graphics Run-Time Libraries. See "Distributing Stand-Alone Applications" on page 1-36 for more information.

2

Fundamentals

This section introduces the MATLAB C++ Math Library. Once you've read this chapter, you'll understand how MATLAB and the MATLAB C++ Math Library work and understand the most important differences between the MATLAB and C++ programming languages.

The section includes these topics:

- "MATLAB Basics" on page 2-3
- "MATLAB for C++ Programmers" on page 2-8
- "C++ for MATLAB Users" on page 2-10
- "Stand-Alone Programs" on page 2-18
- "Learning More" on page 2-24

This overview contains just enough information to get you started. See "Building C++ Applications" in Chapter 1 to learn how to build a simple C++ application. Subsequent chapters contain the examples and the details that this chapter omits.

MATLAB Basics

This section contains an overview of the MATLAB language and programming environment. It does not substitute for a thorough reading of *Using MATLAB*, but it does describe most of the basic concepts in MATLAB. Most of the material in this section, and in this chapter, is repeated and elaborated on in subsequent chapters.

This section describes the interpreted MATLAB environment, not the MATLAB C++ Math Library. Understanding the material in this section will help you understand why the MATLAB C++ Math Library works the way it does.

MATLAB's central data type is the array. Using the MATLAB interpreter, you can create array variables, form arithmetic expressions with arrays, and call functions on arrays. In addition, you can print arrays to and read arrays from files and the screen.

Most of the built-in MATLAB functions and operators are *vectorized*, that is, they operate on entire arrays. For example, to add one array to another, instead of writing a doubly nested `for`-loop, as you would in C or Fortran, you simply call MATLAB's + operator. Similarly, to compute the square root of all the elements in an array, don't loop through the array elements individually calling `sqrt()` on each one. Instead, call `sqrt()` on the entire array; `sqrt()` loops for you.

MATLAB programs consist of a collection of functions. Each MATLAB file can store one or more functions; the filename must end with the extension `.m` (hence the name *M-files*). The primary function in each M-file must have the same name as the file itself.

Note This section concentrates on the MATLAB features supported by the MATLAB C++ Math Library. Some features of the MATLAB language are not yet supported.

Data Types

There are six fundamental data types (classes) in MATLAB, each one a multidimensional array. The six classes are `double`, `char`, `sparse`, `int8`, `cell`, and `struct`. The two-dimensional versions of these arrays are called matrices, hence the name MATLAB. The double precision matrix (`double`) and the character array (`char`) are the data types that are used most frequently. The other data types are for specialized situations like large-scale programming.

Note The MATLAB C++ Math Library does *not* support the `int8` data type.

In MATLAB, every piece of data is an array. For example, a number like 17, which you might think is an integer, is stored by MATLAB as 1-by-1 array containing a double-precision floating-point number.

Every MATLAB array has some basic attributes: the size and shape of the array (i.e., the number of rows, columns, and pages for multidimensional arrays) and the array of double-precision floating-point numbers, which contains the data in the matrix. The data in an array can be either real or complex numbers. If an array stores complex numbers, it acquires a fourth attribute, a second double-precision array of floating-point numbers, the same size as the first. This second array stores the complex part of the matrix data. If an array has zero rows or columns it is a null, or empty, matrix.

Almost every operation or function call in MATLAB creates a new array. There are too many ways to create an array to list them all here. See Chapter 3, “Working with MATLAB Arrays” for a systematic discussion of this topic.

MATLAB also supports string arrays. Each character is 16 bits long; the array prints as strings of characters. To create a string array, surround the string with single quotes, for example, `'This is a string array'`.

Operators

MATLAB supports relational and arithmetic operators. Relational operators typically perform some type of comparison between their two operands, both of which must be the same size, and return an array of ones and zeros the same size as the input arrays. A one in the result array indicates the relationship between the corresponding elements of the input arrays is true, while a zero indicates that relationship is false. The result of a relational operation is always a logical array, an array consisting entirely of ones and zeros.

Arithmetic statements in MATLAB look much like arithmetic expressions in mathematics textbooks or programming languages like C or Fortran. For example, to multiply two arrays, X and Z, and store the result in a third array Y, write Y = X * Z. Note that Y does not necessarily have to exist (but that X and Z must) before this expression is executed.

MATLAB supports the familiar arithmetic operators, +, *, -, / as well as several others, including ^ (exponentiation) and ' (transpose). There are two broad types of arithmetic operators in MATLAB, *array* operators and *matrix* operators.

The array operators are two character operators (except for + and -); the first character is always a . (period). Array operators treat the elements of their array operands individually. For example, C = A .* B represents elementwise, rather than matrix, multiplication of A and B. Each element of the result, C, is the product of the corresponding elements of A and B, that is, C[i] = A[i] * B[i].

Matrix operators are less uniform. There is no single simple formula that describes the behavior of all the matrix operators in MATLAB.

Three of the most useful MATLAB operators are :, (), and []. The first special operator, :, has two different meanings. : permits the generation of sequences of numbers and acts as a wildcard in array subscripts. For example, the expression 1:10 expands into the sequence 1 2 3 4 5 6 7 8 9 10.

The second special operator, (), for array indexing, works closely with :. A simple array subscript, for example, A(3,9), returns the element at the intersection of row three and column nine in the array A. If you want more than a single element, use the : wildcard operator. For example, the expression A(3, :) returns a vector (1-by-N) of all the elements in the third row of the array A.

The third of the three special operators, [], concatenates arrays, either vertically or horizontally. For example, [1 2 ; 3 4] horizontally concatenates 1 and 2 into a vector, 3 and 4 into another vector, and then vertically concatenates the two vectors into a square array.

```
1 2
3 4
```

Within the [] operator, spaces or commas indicate horizontal concatenation, and the semicolon, vertical concatenation.

Functions

In addition to the operators defined by the language, MATLAB ships with a large collection of functions. While there is no way for you to add a new operator to the language, you may add as many functions as you want to MATLAB.

A MATLAB function can take zero, one, or more input arguments, and return zero, one, or more output arguments. In general, a MATLAB function call looks like this:

```
[ x, y ] = foo(a, b, c);
```

This calls the function `foo()` with three input arguments, `a`, `b`, and `c`, and assigns the two results to the output arguments `x` and `y`.

You can also compose functions:

```
x = bar(foo(a, b, c))
```

Note that there is no way to pass multiple return values from one function to the next. In this example, only the first of `foo()`'s return values is passed to `bar()`.

Finally, one or more of a MATLAB function's input or output arguments may be optional. A MATLAB function can never be called with more input or output arguments than it is declared with, but it can always be called with fewer. It is up to the function implementer to put in any necessary error checking.

See "How to Call C++ Library Functions" in Chapter 5 for a complete description of the rules that govern function calls in MATLAB.

Input and Output

MATLAB supports several functions for input and output. The simplest one is `disp()`, short for display. Pass `disp()` an array, and the array appears on the terminal screen. For example:

```
disp('Hello World');
```

Here, `'Hello World'` is a string array.

One group of I/O functions in MATLAB are like their namesakes in the C programming language. MATLAB supports `fprintf()`, `sprintf()`, `scanf()`, and `sscanf()`. `fprintf()` prints to files, `sprintf()` to strings, while `scanf()`

and `sscanf()` read from files and strings, respectively. The arguments to these functions can be simple or complex. For example:

```
sprintf('The answer is: %f\n', magic(2));
```

This call creates this string array:

```
The answer is: 1.000000
The answer is: 4.000000
The answer is: 3.000000
The answer is: 2.000000
```

Notice how the `sprintf()` command recycled its format string argument through the four elements of its data argument.

The I/O functions `load()` and `save()` allow you to save array variables from your application to what's called a MAT-file. That data can then be loaded back in by your application or by another application.

See "Example – Using load() and save() (ex7.cpp)" and "Example – Using File I/O Functions (ex6.cpp)" in Chapter 8 for more details on MATLAB input and output.

Errors

In general MATLAB notifies you of errors by emitting a beep and returning you to the MATLAB prompt. If you need to do more sophisticated error handling in your M-files, you can use a try and catch block to change the flow of control when an error occurs, perform any cleanup, and exit or continue your program. You can also design a method of your own for handling errors and implement it.

Flow of Control

The MATLAB programming language provides an `if`-statement and a `switch`-statement for making decisions and two loop constructs, the `for` loop and the `while` loop, for program iteration. Each of these statements begins a program block (the body of the loop or `if`-statement); you must end the block with the `end` keyword.

See "Control Structure" on page 9-6 for more details.

MATLAB for C++ Programmers

If you're a C++ programmer who has never used MATLAB, make sure you understand the previous section before reading this one. The MATLAB language isn't complicated, but there are important differences between it and C++. These differences won't affect your use of this product directly, because you will, after all, be using C++, but if you understand the differences, you will have a better understanding of the constraints that guided the design of the MATLAB C++ Math Library.

The major differences between MATLAB and C++ include:

- Every MATLAB data object is an array. The two-dimensional version of an array is called a matrix.
- Array objects have value semantics. Think of assignment as copying.
- All functions have call-by-value semantics.
- MATLAB functions can return multiple values.
- MATLAB functions are vectorized.
- In general, MATLAB functions do *not* have side effects; they do not modify their inputs.
- In MATLAB, subscripts begin at 1 rather than 0.
- MATLAB arrays store data in column-major, rather than C++'s row-major, order.
- Memory management is handled by the MATLAB interpreter.

MATLAB is a more specialized programming language than C++ and, as a result, lacks much of the machinery of C++, such as typed variables and name space management. MATLAB is not a general-purpose programming language like C++, so it is not nearly as versatile as C++. However, in its domain, numerical linear algebra, MATLAB is far easier to use and much more concise than C++. This library brings some of that power to C++ programmers.

Much of MATLAB's expressive power stems from its rich collection of numerical operators. In C++ it is impossible to emulate perfectly MATLAB's operator syntax, because some of the MATLAB operators like `.*` consist of two characters, while others like `'` are not legal C++ operators. However, many of MATLAB's operators are present in C++, overloaded to provide commutativity and inlined for efficiency. Those MATLAB operators that are not present as C++ operators are available as function calls.

Fortunately, one of MATLAB's most powerful operators, `()`, for array indexing, is a valid C++ operator. The indexing operator can access a single element or a group of elements in an array. For example, in MATLAB, `A(2:4, 1:3)` returns a 3-by-3 array consisting of the second, third and fourth elements in the first three columns of array `A`. Because the `:` is not a valid C++ operator, the equivalent expression in C++ requires the use of the `colon()` function: `A(colon(2,4), colon(1,3))`. The indexing operator is also the only operator that can modify an array. For example, the expression

```
A(4,7) = 13
```

writes the value `13` into the entry at row 4 and column 7 of array `A`. This is the only way to modify the contents of an array. Because the MATLAB cell array indexing operator, `{}`, is not a valid C++ operator, the MATLAB C++ Math Library uses the `cellhcat()` routine to emulate it.

You have seen that an array subscript can itself be an array. When an array subscript is a logical array containing only zeros and ones, it is called a *logical* index. A logical index acts like a mask or filter. Each element in the logical index corresponds to an element in the subscripted array. If the element in the logical index is 1, the corresponding element in the subscripted array appears in the result. The result of any nontrivial (array of all ones or all zeros) logical indexing operation can have various shapes depending on the shape of the indices.

See "Differences Between C++ and MATLAB" on page 9-2 for a more comprehensive list of differences between MATLAB and C++. See "MATLAB C++ Math Library Basics" on page 2-12 to learn how these principles are expressed in the MATLAB C++ Math Library.

C++ for MATLAB Users

The MATLAB C++ Math Library should hold very few surprises for the average MATLAB user, as it was designed to be as similar to MATLAB as possible. There are however, a few areas of difference, mostly due to immutable features of C++.

How the Library Is Similar to MATLAB

- All the variables are arrays.
- Most of the mathematical operators (`+`, `*`, `/`, `-` and others) are available.
- The indexing operator, `()`, works just as it does in MATLAB.
- The MATLAB C++ Math Library manages memory for you.

Though this list is short, it represents a substantial similarity between MATLAB and the library. Many MATLAB expressions translate verbatim into C++. The fundamental goal of MATLAB and the MATLAB C++ Math Library is the same: to provide an expression-oriented programming environment for the development of numerical linear algebraic algorithms.

How C++ and the Library Differ from MATLAB

- The syntax is slightly different; for example, there is no `:` operator or cell array `{ }` indexing operator.
- Functions in C++ can return only one value.
- C++ flow-of-control statements (`if`, `for`, and `while`) are different from their MATLAB equivalents.
- C++ supports pointers and references.
- You must declare variables before using them.
- C++ uses exceptions to report errors.

The most obvious difference between C++ and MATLAB is the syntax. Many of MATLAB's neat syntactic tricks, such as `:` and `'`, are not valid C++ syntax, and therefore are available only through function calls in C++. Though the lack of operators may at first seem to be the biggest difference, it is not. Since each of the operators has an equivalent function, the only thing missing is convenience of syntax.

A much bigger difference is the lack of multiple return values in C++. A C++ function returns either zero or one value. To simulate MATLAB's multiple return values, the MATLAB C++ Math Library requires that you pass all but the first of your return values as inputs to the function; you must pass these arguments as pointers to arrays so that the called function can modify them.

The differences between the flow of control statements (`if`-statements, `for`- and `while`-loops) in MATLAB and C++ are minor. Certainly the syntax is a bit different, but the most important difference is that the arguments to these statements in C++ must be scalars. Because `if` and `while` require Boolean values, you must use the MATLAB C++ Math Library `tobool()` function to reduce any array that appears in one of these statements to a scalar. `tobool()` returns a Boolean result.

Variable declaration, required by C++, is another difference, but a minor one, especially since C++ allows you to declare variables at any point in the program. Also, the C++ compiler will forcefully remind you of any variables you have forgotten to declare.

C++ and MATLAB both support `try` and `catch` blocks, which allow you to detect and recover from an error. However, C++'s exception-handling mechanism is somewhat more comprehensive than MATLAB's error handling because you can associate an object with a `catch` block. In MATLAB, a `catch` block catches any error.

A C++ exception is an object created when an error occurs. The exception typically identifies the type of error and its location. Like a MATLAB error, an unhandled C++ exception will terminate the program. Unlike a MATLAB error, you won't get to see the error message associated with the exception unless you catch the exception and print it out manually — C++ will *not* do this for you automatically.

See "Differences Between C++ and MATLAB" in Chapter 9 for more details on the differences between C++ and MATLAB.

MATLAB C++ Math Library Basics

This section contains an overview of the MATLAB C++ Math Library. It provides an introduction to most of the basic concepts in the library. Because of its brevity, it does not discuss each concept in great detail; subsequent chapters provide much more depth. Before you read this section, make sure you understand all of the concepts in "MATLAB Basics" on page 2-3.

Like MATLAB, the MATLAB C++ Math Library's central data type is the array. Using the MATLAB C++ Math Library, you can create array variables, form arithmetic expressions with arrays, and call functions on arrays. In addition, you can print an array to a file or display it on the screen, or read an array from a file or from the screen.

Most of the built-in MATLAB functions and operators are *vectorized*, that is, they operate on an entire array. This is true of the routines in the MATLAB C++ Math Library as well. For example, to add one array to another, instead of writing a doubly nested `for`-loop, as you would in C or Fortran, simply call the library's + operator. Similarly, to compute the square root of all the elements in an array, don't loop through the array elements individually calling `sqrt()` on each one. Instead, call `sqrt()` on the entire array; `sqrt()` will loop for you.

Data Types

Arrays are represented by objects of the class `mwArray`. However, unlike MATLAB, C++ allows numbers like 17 to be declared integers rather than 1-by-1 arrays, at a considerable savings in space and increased speed. All the routines in the MATLAB C++ Math Library can handle integers, double-precision floating-point numbers, or strings as easily as arrays. C++ automatically converts the scalars or strings into arrays before the routines are called.

Every `mwArray` class object contains a pointer to a MATLAB array structure. For this reason, the attributes of an `mwArray` object are a superset of the attributes of a MATLAB array. Every MATLAB array contains information about the size and shape of the array (i.e., the number of rows, columns, and pages) and either one or two arrays of data. The first array stores the real part of the array data and the second array stores the imaginary part. For arrays with no imaginary part, the second array is not present. The data in the array is arranged in column-major, rather than row-major, order.

The mwArray class has a small interface. Most of the functions in the MATLAB C++ Math Library are not members of the mwArray class. Having a small interface means that mwArray is an easy class to understand and one that is less likely to change as the library grows. Chapter 10 describes the interface completely.

The MATLAB C++ Math Library defines two other important data types: mwIndex and mwSubArray. Both of these classes are used by the array indexing routines. mwIndex objects represent the index applied to the array and mwSubArray objects represent the subscripting operation itself. Casual users of the library won't need to use these two classes. See "The mwIndex Class" in Chapter 4 for complete details.

You can make an mwArray from any number of other data types: integers, double-precision floating-point real numbers, strings (delimit C++ strings with "" rather than ''), an instance of an mwIndex or mwSubArray. Like the routines in MATLAB, most of the operators and function calls in the MATLAB C++ Math Library return a newly allocated array.

Operators

Operator syntax is a very convenient and natural looking shorthand for function calls. The MATLAB C++ Math Library supports a *subset* of the operators available in MATLAB. The library provides all of the relational operators and those arithmetic and miscellaneous operators that do not violate rules of C++ syntax. The operators that are not available as operators are available via function calls.

In MATLAB there are two classes of arithmetic operators: array operators and matrix operators. In the MATLAB C++ Math Library the arithmetic operators are *matrix* operators, except for + and - for which the distinction is meaningless. This means that, for example, A * B is the linear algebraic product (matrix multiplication) of A and B, rather than the elementwise product of A and B. A and B are mwArray objects. All of the arithmetic array operators are also available via function calls.

Operators in the MATLAB C++ Math Library are vectorized. This means you can use the + operator, for example, to compute the sum of two arrays without using a loop. In C++, without some kind of an array class, you'd use one or two for-loops to compute the sum of two arrays; e.g., for every row in the array and for every column in the row, compute the sum at the row/column intersection.

The operators in the MATLAB C++ Math Library all contain loops of this sort already, so there is no need for you to write them.

The section “Operators” in Chapter 11 lists the available operators and their function call equivalents.

Functions

The MATLAB C++ Math Library contains over 400 mathematical functions and a collection of utility routines. The mathematical functions are C++ versions of their MATLAB counterparts, while the utility routines provide services that the mathematical functions previously received from interpreted MATLAB; for example, printing and memory management.

Unlike C++ functions, MATLAB functions may have multiple return values. The MATLAB C++ Math Library provides for multiple return values by requiring that you pass all your return values except the first into the function as output parameters. In a function argument list, output parameters always precede input parameters. For example, the MATLAB function call `[ V, D ] = eig(X)` becomes `V = eig(&D, X)` in the MATLAB C++ Math Library.

By default all MATLAB functions have optional input and output arguments (see “Functions” on page 2-6). However, for each function, only certain combinations of input and output arguments are valid. The MATLAB C++ Math Library uses a combination of function overloading and C++ default arguments to make available for each function those exact combinations of input and output arguments that are valid in MATLAB. For example, the MATLAB function `svd()` has a maximum of two inputs and three outputs, a total of 12 different ways it might be called. However, only three of those combinations are valid. There are, therefore, three versions of `svd()` in the MATLAB C++ Math Library.

See “How to Call C++ Library Functions” in Chapter 5 for a more thorough explanation of how to determine what arguments to pass to the MATLAB C++ Math Library version of a MATLAB function call. You can also use the online *MATLAB C++ Math Library Reference* available from the Help Desk to find the specific arguments for each library function. “Accessing Online Reference Documentation” on page 1-7 describes how to access the Help Desk.

Input and Output

MATLAB programs use `scanf()` and `fprintf()` to read and write from input and output and `load()` and `save()` to read and write array variables from and to MAT-files. C++ introduces a new concept: input and output streams. The MATLAB C++ Math Library supports MATLAB's `fscanf()` and `fprintf()` style of input and output along with `load()` and `save()`, and also provides the necessary operators for C++ stream input and output.

The MATLAB and MATLAB C++ Math Library versions of the `fprintf()` and `fscanf()` style functions are essentially the same. See "Example – Using File I/O Functions (ex6.cpp)" in Chapter 8 for more information on the input and output functions. The MATLAB C++ Math Library versions of `load()` and `save()` allow you to share data with MATLAB applications or with other applications developed with the MATLAB C++ or C Math Library; however they do not provide as many options as the MATLAB versions. See "Importing and Exporting MAT-File Data" in Chapter 8 to learn how the functions differ.

In many ways, streams are more convenient than functions like `fprintf()`, because they are more consistent, flexible, and extensible. There are two basic types of streams, input streams and output streams. A C++ stream is a sequence of data objects. Often a stream consists of a sequence of characters. Streams can be attached to one of many types of data sources, or sinks: files, strings, and the screen, for example.

Each object in a C++ program is responsible for printing itself to a stream and reading itself from a stream. This decentralizes the responsibility for input and output formats, which means objects have complete control over their own printed format, and new objects can be added without changing the code in the basic streams mechanism. Furthermore, since the interface to each type of stream is the same, the code to save an object into a file is identical to that used to print that object on the screen or send it over the network to another process.

C++ defines three standard streams, `cin`, `cout`, and `cerr`. `cin` is bound to standard input, `cout` to standard output and `cerr` to standard error. To send an array `A` to the standard output, you write:

```
cout << A << endl;
```

To read an array in from standard input, you write:

```
cin >> A;
```

To send an array `A` to standard error, you write:

```
cerr << A << endl;
```

<< is the output operator and >> the input operator. The direction in which the operator points suggests the direction in which data flows. “Using Array Stream I/O” in Chapter 8 describes C++ stream-style output and the array I/O format completely. Refer to that section for more information.

Errors

The MATLAB C++ Math Library uses C++ exceptions to report errors. The MATLAB C++ Math Library divides the errors it reports into categories and for each of these categories it provides a class. All of the exception classes are subclasses of `mwException`. Because all the exceptions are derived from the same superclass, it is easy to write a general exception handler. For example:

```
try {
    // Some MATLAB C++ Math Library code
}

catch(mwException &ex) {
    cout << ex << endl;
}
```

This try-catch block catches any exception that occurs during the execution of the indicated MATLAB C++ Math Library code and prints the error message associated with the exception to standard output. You should put a try-catch block like this one in every `main()` routine you write.

See “Handling Exceptions” in Chapter 7 for more information on the error handling mechanism and Appendix C for a list of the library’s error messages.

Memory Management

MATLAB users usually don’t worry about memory management because the MATLAB interpreter manages memory for them. This is in marked contrast to most programming languages, which require their users to explicitly manage their own memory. The MATLAB C++ Math Library uses a memory management scheme that both performs well and ensures there are no memory leaks. This means that, in most cases, users of the MATLAB C++ Math Library do not need to implement complex memory management mechanisms because the library already contains one.

If you need to change the way the library allocates its memory, the library provides memory management routines that let you substitute your own scheme. “Memory Management” in Chapter 7 describes how to use the routines.

Stand-Alone Programs

In addition to writing M-files, there are three other ways you can call MATLAB functions: via MEX-files or via the MATLAB Engine, or by using either the MATLAB C or C++ Math Library. Any M-file, MEX file, or Engine code you write requires the entire MATLAB environment to run. However, with the MATLAB C and C++ Math libraries, you can write stand-alone (external) programs.

A stand-alone program offers several advantages:

- It is often faster than the equivalent interpreted MATLAB program.
- It is generally smaller in executable size and requires less memory than the same program written as an M-file.
- It can be redistributed to your customers, even if those customers don't own MATLAB. See "Building a Stand-Alone Application on PCs" on page 1-23 and "Distributing Stand-Alone Applications" on page 1-36.

However, there are disadvantages to stand-alone programs:

- You can't use the MATLAB functions `eval()` or `input()`.
- You can't call a Handle Graphics® function.
- Certain parts of MATLAB syntax, for example, `:` and `[]`, are not available in C or C++.
- You can't call functions in the MATLAB toolboxes.
- You have no access to Simulink®.

Stand-alone programs are best suited for highly numeric applications. You can, of course, incorporate calls to third-party libraries, such as the X Window System, the Microsoft Windows Graphical Device Interface or MFC, in your stand-alone programs.

You can also use the MATLAB C++ Math Library to develop one or more modules or parts of a larger program. For example, you may have a signal processing application for which you want to do algorithm development in MATLAB. To do this, you write M-files that solve your signal processing problems. Using the MATLAB C++ Math Library, you can quickly translate these M-files into C++. Then you plug the resulting C++ code into your larger program. The translation will be even faster if you use the MATLAB Compiler, which is sold separately, to automatically translate M-files to C++.

By using interpreted MATLAB for algorithm development and rapid prototyping, the MATLAB Compiler for translation to C++, the MATLAB C++ Math Library to enable the construction of external modules, and C++ for the larger program framework, you use the strengths of each.

Example Program: Writing Simple Functions (ex4.cpp)

This example demonstrates how to write a simple function that takes two matrix arguments and returns a matrix value. You can find the code for this example in the `<matlab>/extern/examples/cppmath` directory on UNIX systems and in the `<matlab>\extern\examples\cppmath` directory on PCs, where `<matlab>` represents the top-level directory of your installation. See “Building C++ Applications” on page 1-13 for information about building and running the example program.

In the example, note the following:

- Your routines should return an `mwArray` object, *not* a reference to one.
- `mwArray` objects are most efficiently passed by reference.
- Input arrays should be declared `const`.
- The vectorized routines in the MATLAB C++ Math Library eliminate, in many cases, the need for you to write explicit `for`-loops in your own code.

```
    // ex4.cpp

    #include <stdlib.h>
(1) #include "matlab.hpp" // <1>

(2) static double data0[] = { 2, 6, 4, 8 }; // <2>
    static double data1[] = { 1, 5, 3, 7 };

(3) mwArray average(const mwArray &m1, const mwArray &m2) // <3>
    {
        return rdivide(plus(m1, m2), 2);      // (m1 + m2) / 2
    }

(4) int main(void) // <4>
    {
        // Create two matrices
(5)     mwArray mat0(2, 2, data0);    // mat0 = [ 2 4; 6 8 ] <5>
        mwArray mat1(2, 2, data1);    // mat1 = [ 1 3; 5 7 ]
        mwArray mat2;

(6)     mat2 = average(mat0, mat1); // <6>

(7)     cout << mat0 << "\t + \n" << mat1
             << "\t / 2 = \n" << mat2; // <7>

        return(EXIT_SUCCESS);
    }
```

The numbered items in the list below correspond to the numbered sections of code example:

1. Include header files. `matlab.hpp` declares the MATLAB C++ Math Library's data types and functions. `matlab.hpp` includes `iostream.h`, which declares the input and output streams `cin` and `cout`. `stdlib.h` contains the definition of `EXIT_SUCCESS`.

2. Declare the data that is used to initialize the arrays in the main program. As noted in "Example Program: Creating Arrays and Array I/O (ex1.cpp)" on page 3-15, use a one-dimensional C++ array to initialize a Math Library

array from C++ static data; the `mwArray` constructor takes a one-dimensional array as an argument.

Remember that C++ stores its two-dimensional arrays in row-major order, whereas the C++ Math Library stores arrays in column-major order. When setting up static matrix data, always enter the data in column-major order.

3 Declare the `average()` function, which "averages" two matrices. For each pair of elements in the two input matrices, `m1(i,j)` and `m2(i,j)`, the function computes the average of the two elements and stores the result in the corresponding element of the output array.

For efficiency, pass the two input matrices by `const` reference. The `const` indicates that the input parameters are not modified.

`average()` returns an `mwArray` rather than a reference to an `mwArray`. Your functions should never return a reference to an `mwArray` because returning a reference to a local variable is an error in C++. Refer to a C++ reference guide for more information.

Note that this routine does not contain an explicit loop. The functions you call – `rdivide()` and `plus()` – contain the necessary loops. Vectorized functions like these are common in the MATLAB C++ Math Library and provide a great convenience: you don't need to write the loops to process array data.

4 Declare the main routine. `main()` declares the matrix variables, calls the `average()` function, and prints the results.

5 Declare three matrices. Initialize `mat0` and `mat1` to 2-by-2 square matrices, using the static data declared earlier. The data initializing `mat0` and `mat1` is arranged in column-major order. The first row of `mat0` is `2 4`, the second `6 8`. The first row of `mat1` is `1 3`, the second `5 7`. Without a specified size or initial data, `mat2` is a null or empty array.

6 Call the `average()` function. Pass `mat0` and `mat1` as input arguments and assign the result to `mat2`.

7 Print the result. This code demonstrates another convenience of C++: a single line of code sending more than one object to an output stream. The output appears on the stream in the same order it appears in the code.

Output

The program produces this output:

```
[
       2        4 ;
       6        8
]

       +
[
       1        3 ;
       5        7
]

       / 2 =
[
  1.50000   3.50000 ;
  5.50000   7.50000
]
```

Writing Efficient Programs

The general rule for writing efficient programs with this library is to use scalars wherever possible.

Operations on integers and doubles are at least one order of magnitude faster than the corresponding operations on arrays. The use of scalars has the most impact in indexing and arithmetic expressions. Wherever possible, use integers instead of 1-by-1 arrays in indexing expressions, and doubles rather than 1-by-1 arrays in arithmetic expressions.

However, do not let the preceding comments discourage you from using the full power of the interface. Using the efficiency of scalars helps your code run faster, but you should not base your designs on it. Your design and development time are worth much more than a few CPU-cycles.

The table below demonstrates several cases where you can use doubles and integers to improve the efficiency of your programs.

Table 2-1: Using Scalars for Efficiency

MATLAB code	Naive C++ Translation	Efficient C++ Translation	Reasons

`C = A(3) * B(4);`	`mwArray A, B, C;`
	`C = A(3) * B(4);`
		`double C;`
		`mwArray A, B;`
		`C = A(3) * B(4);`
			Use of double as result.

`n = max(size(A))`
`A(n) = n*n;`
	`mwArray n, A;`
	`n = max(size(A));`
	`A(n) = n*n;`
		`int n;`
		`mwArray A;`
		`n = max(size(A));`
		`A(n) = n*n;`
			Use of integer as index. Integer rather than matrix multiplication.

Learning More

This short chapter doesn't cover all the details of MATLAB and the MATLAB C++ Math Library. To help you navigate through the rest of this document, here is a list of the topics discussed in this chapter and references to help you find further information:

- **Arrays and Matrices**
  "Overview" in Chapter 3
  "Performing Common Array Programming Tasks" in Chapter 3
  "Example Program: Creating Arrays and Array I/O (ex1.cpp)" in Chapter 3
- **Indexing or Subscripting**
  "Indexing into Arrays" in Chapter 4
  "Duplicating a Row or Column" in Chapter 4
- **Calling Functions**
  "How to Call C++ Library Functions" in Chapter 5
  "Example Program: Calling Library Functions (ex2.cpp)" in Chapter 5
- **Operators**
  "Overview" in Chapter 6
  "Operators" in Chapter 11
- **Input and Output**
  "Example – Using load() and save() (ex7.cpp)" in Chapter 8
  "Example – Using File I/O Functions (ex6.cpp)" on page 8-15
  "Using Array Stream I/O" in Chapter 8
  "Using File I/O Functions" in Chapter 8
  "Importing and Exporting MAT-File Data" in Chapter 8
- **Errors**
  "Example Program: Handling Exceptions (ex5.cpp)" in Chapter 7
  "Exception Handling in the MATLAB C++ Math Library" in Chapter 7
- **Syntax**
  "Differences Between C++ and MATLAB" in Chapter 9
  "Example Program: Rewriting roots.m in C++ (ex8.cpp)" in Chapter 9

If your question isn't answered in this document, there are several other places you can go for help:

- Other reference books:

  MATLAB C Math Library User's Guide

  MATLAB Application Program Interface Guide

  Online *MATLAB Function Reference*

  Using MATLAB
- The MathWorks Technical Support on our home page:

  `http://www.mathworks.com`
- The MathWorks Technical Support Solution Search Engine at:

  `http://www.mathworks.com/solution.html`
- The MATLAB Usenet newsgroup, `comp.soft-sys.MATLAB`.
- MATLAB Technical Support e-mail address: `support@mathworks.com`.
- The MATLAB Technical Support phone center:

  (508) 647-7000 (voice)

  (508) 647-7201 (fax)
- U.S. Mail:

  The MathWorks, Inc.

  3 Apple Hill Drive

  Natick, MA 01760-2098

3

Working with MATLAB Arrays

Overview

To use the routines in the MATLAB C++ Math Library, you must pass your data to the routines in the form of a MATLAB array object. This chapter:

- Describes the MATLAB arrays supported by the library and the C++ object defined to represent them.
- Describes how to create arrays of all types and perform other common array programming tasks.

Because the library routines work the same as the corresponding MATLAB functions, this chapter does not describe their function in detail. For more information about MATLAB arrays and their use, see *Using MATLAB*. Instead, this chapter provides an overview of working with MATLAB arrays and highlights where the syntax of the library routine is significantly different than its MATLAB counterpart.

Supported MATLAB Array Types

The MATLAB C++ Math Library supports the following MATLAB array types (or classes).

- Numeric arrays—The library supports multidimensional numeric arrays, where values are represented in double precision format. All MATLAB arithmetic functions operate on numeric arrays. For more information, see “Numeric Arrays” on page 3-4.
- Sparse arrays—To conserve space, two-dimensional numeric arrays can be stored in sparse format, where only nonzero elements of the array are stored. Numeric arrays with more than two dimensions cannot be converted to sparse format. For more information, see “Sparse Matrices” on page 3-19.
- Character arrays—The library supports multidimensional arrays of characters, represented in 16-bit ASCII Unicode format. For more information, see “Character Arrays” on page 3-24.
- Cell arrays—The library supports multidimensional arrays of MATLAB’s primary container type called *cells*. Each cell can contain any type of MATLAB array, including other cell arrays. For more information, see “Cell Arrays” on page 3-28.
- Structures—The library supports multidimensional arrays of MATLAB’s other container type called *structures*. A structure can be thought of as a

one-dimensional cell array where each cell is assigned a name. These named cells, called *fields*, define the organization of the structure. Do not confuse MATLAB structures with standard C structures. For more information, see “MATLAB Structures” on page 3-35.

Choose the MATLAB array type that best fits your data. For more detailed information about these array types, see *Using MATLAB*.

MATLAB Array C++ Object

The MATLAB C++ Math Library uses one object (or class), `mwArray`, to represent all types of MATLAB arrays. Each instance of this object contains information including the type of MATLAB array and its size and shape. The object also contains the data stored in the array. This class, like any other C++ classes, defines a set of constructors. Whenever you create a variable of this type, one of these constructors is called.

Note Do not confuse the `mwArray` object with the `mxArray` data type supported by the MATLAB C Math Library.

Numeric Arrays

The MATLAB C++ Math Library includes routines to create and manipulate numeric arrays. Numeric arrays are the fundamental MATLAB array type. MATLAB supports other numeric array types, such as uint8; however, these data types are only used for importing and exporting image data.

The following table lists the MATLAB C++ Math Library routines to create numeric arrays and perform some basic tasks with them. The sections that follow provide more detail about using these routines. For more detailed information about using numeric arrays, see *Using MATLAB*. For more detailed information about any of the library routines, see the online *MATLAB C++ Math Library Reference*.

Table 3-1: Numeric Array Routines

To ...	Use ...
Create an uninitialized array.	mwArray default constructor: `mwArray A;`
Create an empty ([]) array.	`empty()`
Create an initialized scalar (1-by-1) array from a double precision floating point number.	mwArray scalar constructor: `mwArray(double)`
Create an initialized scalar (1-by-1) array from an integer.	mwArray scalar constructor: `mwArray(int)`

Create an initialized *m*-by-*n* array (matrix) from double, integer, or unsigned short data.	mwArray matrix constructors:
	`mwArray(int, int, double*, double*)`
	`mwArray(int, int, int*, int*)`
	`mwArray(int, int, unsigned short*, unsigned short*)`

Copy an existing mwArray object.	mwArray copy constructor: `mwArray( mxArray *)`

Table 3-1: Numeric Array Routines (Continued)

To ...	Use ...
Copy an existing `mxArray` data type (returned by a MATLAB C Math Library routine or MATLAB API routine.)	`mwArray` copy constructor: `mwArray(const mwArray&)`
Create a *1*-by-*n* integer ramp.	`mwArray` ramp constructor: `mwArray(int, int, int)`
Create an `mwArray` from a subarray (used in indexing.)	`mwArray` subarray constructor: `mwArray(const mwSubArray&)`
Create an *m*-by-*n* array by concatenating existing arrays	`horzcat()` `vertcat()`
Create an array with more than two dimensions (*m*-by-*n*-by-*p*-by...)	`cat()` or by using assignment

Create an array with more than two dimensions (*m*-by-*n*-by-*p*-by...) of ones, zeros, or random numbers.	`ones()`
	`zeros()`
	`rand(), randn()`

Create an identity matrix or magic square.	`eye()` `magic()`

Creating Numeric Arrays

You can create a numeric array in a C++ program by:

- Using an `mwArray` constructor
- Using an array creation routine
- Calling an arithmetic routine
- Concatenating existing arrays
- Assigning a value to an element in an array

The following sections provide more detail about using each of these mechanisms, highlighting areas where the C++ syntax is significantly different from the corresponding MATLAB syntax.

Creating Arrays with C++ Constructors

As a C++ class, the mwArray interface includes many useful constructors that allow you to create many different types of array. There is no MATLAB equivalent for a constructor.

When you declare a mwArray object, as in the following

```
mwArray A;
```

you invoke the default constructor which creates a uninitialized array.

Note Do not pass an uninitialized array to a MATLAB C++ Math Library routine. Assign it a value before passing it to a routine.

The mwArray object supports other constructors that accept various combination of arguments that allow you to create numerical scalar arrays or copy an existing array. (For a complete list of all mwArray constructors, see “Constructors” in Chapter 10.)

The following example uses the mwArray matrix constructor to create an 2-by-3 matrix, initialized to the values in a C++ array of double precision values. You can also use C++ arrays of integers or unsigned short values to initialize a MATLAB mwArray. This constructor can optionally take a second C++ array to initialize the imaginary part of an array of complex numbers. The example then uses the mwArray copy constructor to make a copy of the 2-by-3 array.

```
double data[] = {1,4,2,5,3,6};

mwArray C(2, 3, data); // matrix constructor
mwArray D(C);          // make a copy of C

cout << "C =" << C << endl;
cout << "copy of C =" << D << endl;
```

This code produces the following output:

```
C = [
       1     2     3;
       4     5     6
  ]
Copy of C = [
```

```
        1    2    3;
        4    5    6
    ]
```

Creating Multidimensional Arrays with Constructors. You cannot create an array of more than two dimensions using an `mwArray` constructor. Use an array creation routine or concatenation to create multidimensional arrays. Alternately, you can create a two-dimensional array using a constructor and then change it into a multidimensional array using the `reshape()` routine. For more information about the reshape() routine, see the online *MATLAB C++ Math Library Reference*.

Using Array Creation Routines

The MATLAB C++ Math Library provides routines that create commonly used MATLAB arrays, such as arrays of 0's or 1's.

- Array filled with ones, `ones()`
- Array filled with zeros, `zeros()`
- An empty array, equivalent to the MATLAB `[]`, `empty()`
- Identity matrices, `eye()`
- Random numbers, `rand()`
- Normally distributed random numbers, `randn()`
- Magic squares (limited to two-dimensions), `magic()`

When you call these routines, you define the number of dimensions of the array by the number of dimensions you specify as arguments. Given *n* arguments, the routines return a multidimensional array with the *n* dimensions. (The eye() routine and the `magic()` routine only support two dimensional arrays.)

For example, this code fragment creates a 2-by-3-by-2 array of normally distributed random numbers.

```
mwArray B;

B = randn(2,3,2); // Create 3-d array

cout << "B =" << B << endl;
```

This code produces the following array:

```
B =[
(:,:,1) =
   [
   -0.4326    0.1253   -1.1465 ;
   -1.6656    0.2877    1.1909
   ]
(:,:,2) =
   [
    1.1892    0.3273   -0.1867 ;
   -0.0376    0.1746    0.7258
   ]
]
```

Note You can specify a zero value for any dimension. MATLAB considers any array with a zero dimension an empty array.

Creating Integer Ramps. In MATLAB, the : (colon) operator can be used as a fast way to create a vector of monotonically increasing numbers. This capability is often used as a wildcard in MATLAB array indexing expressions. The MATLAB C++ Math Library uses two routines to emulates the : operator:

- `ramp()` to create vectors
- `colon()` to specify a range of values in indexing expressions. Unlike the MATLAB colon operator, the `colon()` routine cannot be used to specify the bounds of a C++ `for`-loop. For more information about using the `colon()` routine in array indexing expressions, see Chapter 4.

When you use `ramp()`, the first argument represents the starting value and the second argument represents the end value. As an example, the following code fragment creates a vector of all the numbers between 1 and 10.

```
mwArray A = ramp(1,10);
```

The library also supports the three-argument form of `ramp()`, where the first argument represents the starting value, the second argument represents the size of the increment between values and the third argument.

Calling MATLAB Arithmetic Routines

As in MATLAB, most of the operators and functions in the MATLAB C++ Math Library create at least one new array as their result. For example, when you multiply two arrays, the result is a new array. This code demonstrates how multiplying a 4-by-4 array of 1's by the 4-by-4 identity matrix creates a new array, C.

```
mwArray A, B;

A = ones(4);
B = eye(4);

mwArray C = A * B;
```

C is a new array; the result of the multiplication.

Using Concatenation

Vertical and horizontal concatenation are useful ways to construct arrays of any size and shape. In MATLAB, the concatenation operator ([]) performs both operations. The MATLAB C++ Math Library uses two routines to emulate this operator:

- horzcat() concatenates arrays horizontally.
- vertcat() concatenates arrays vertically.

Concatenating Horizontally. In MATLAB, you can horizontally concatenate the scalar arrays 1, 2, 3, 4, 5, and 6 into a vector containing one row and six columns.

```
A = [ 1 2 3 4 5 6]

A =

    1 2 3 4 5 6
```

You can create the same vector in C++ code using horzcat().

```
mwArray A;

A = horzcat( 1, 2, 3, 4, 5, 6 );
```

```
cout << “A = “ << A << endl;
```

This code fragment produces this output:

```
A = [
        1    2    3    4    5    6
  ]
```

Concatenating Vertically. To vertically concatenate the same scalar arrays into a 2-by-2 matrix in MATLAB, insert a semicolon in the list of arrays where you want to create rows:

```
A = [ 1 2 3; 4 5 6 ]

A =

    1 2 3
    4 5 6
```

To create this matrix in a C++ program, you must use `vertcat()`, using nested calls to `horzcat()` to create the rows.

```
mwArray A;

A = vertcat( horzcat(1, 2, 3), horzcat(4, 5, 6) );
```

This code fragment produces this output:

```
A = [
        1    2    3 ;
        4    5    6
  ]
```

`horzcat()` and `vertcat()` work on vectors and two-dimensional arrays as well as scalars. For example, the following code fragment concatenates the two dimensional arrays `A` and `B` to create the two-dimensional array `C`.

```
mwArray A = vertcat( horzcat(1, 2, 3), horzcat(4, 5, 6) );
mwArray B = vertcat( horzcat(1, 2, 3), horzcat(4, 5, 6) );
mwArray C = vertcat( A, B );
```

Horizontally concatenated arrays must have the same number of rows; vertically concatenated arrays must have the same number of columns.

Using Concatenation to Create Arrays of More Than Two Dimensions. To create arrays of more than two-dimensions through concatenation, use the `cat()` routine. You cannot create arrays of more than two dimensions using `horzcat()` and `vertcat()`.

In MATLAB, the `cat` function concatenates a group of arrays along a specified dimension using the following syntax

```
B = cat(dim,A1,A2...)
```

where `A1`, `A2`, and so on are the arrays to concatenate, and `dim` is the dimension along which to concatenate the arrays.

For example, this MATLAB code concatenates the two-dimensional arrays `A` and `B` into a three-dimensional array using the `cat` function.

```
A = [ 1 1 1; 2 2 2 ];
B = [ 3 3 3; 4 4 4 ];

C = cat(3, A, B)
C(:,:,1) =
     1     1     1
     2     2     2
C(:,:,2) =
     3     3     3
     4     4     4
```

This code fragment creates the same three-dimensional array in a C++ program:

```
mwArray A = vertcat( horzcat(1, 1, 1), horzcat(2, 2, 2) );
mwArray B = vertcat( horzcat(3, 3, 3), horzcat(4, 4, 4) );
mwArray C = cat( 3, A, B );

cout << "C =" << C << endl;
```

This code produces the following output:

```
C =[
(:,:,1) =
  [
     1     1     1;
     2     2     2
  ]
```

```
(:,:,2) =
  [
     3     3     3;
     4     4     4
  ]
]
```

If the number of dimensions you specify in `dim` is greater than the number of arrays you specify as arguments, `cat` automatically adds subscripts of 1 between dimensions, if necessary. For example, if you change `cat(3,A,B)` to `cat(4,A,B)`, the code produces the following output. Note the added dimension in the index subscripts.

```
C =[
(:,:,1,1) =
  [
     1     1     1 ;
     2     2     2
  ]
(:,:,1,2) =
  [
     3     3     3 ;
     4     4     4
  ]
]
```

Using Assignment

You can create scalar arrays using the C++ assignment (=) operator. For example, the following C++ code creates an array named `A` and assigns the value 5 to A.

```
mwArray A = 5;
```

The result of this assignment is a 1-by-1 array (one row, one column) containing the single number 5.0 represented in double-precision floating-point format.

You can assign a nonscalar value to a variable that contains a scalar array, or a scalar value to a variable that contains a nonscalar array. In both cases, the MATLAB C++ Math Library manages the memory associated with each array to ensure that there are no memory leaks.

You can also create string arrays using the C++ assignment operator. The following C++ code creates an array named A and assigns the value "abcd" to A.

```
mwArray A = "abcd";
```

Creating Multidimensional Arrays By Assignment. You can create multidimensional arrays using indexed assignment statements. You use MATLAB array indexing to specify a location in an array. MATLAB creates the array (or extends an existing array) to accommodate the location specified.

For example, the following assignment statement creates a new three dimensional array by assigning a single value to the location specified by row 2, column 2, page 2. MATLAB fills the elements of the array with zeros before making the assignment.

```
H(2,2,2) = 5

H(:,:,1) =
     0     0
     0     0
H(:,:,2) =
     0     0
     0     5
```

The MATLAB C++ Math Library supports this same syntax. You can create a multidimensional array by assigning a value to a location in the array. The library creates an array (or extends an existing array) to accommodate the location. The following C++ code fragment creates the same three dimensional array.

```
mwArray H;
H(2,2,2) = 5;
```

Note Do not declare an array and perform an indexed assignment in the same statement. The statement `mwArray H(2,2,2)= 5` is not valid.

Initializing a Numeric Array with Data

Using concatenation to build large arrays can become cumbersome. As an alternative, you can put the data into a standard C++ array and pass it to the `mwArray` matrix constructor, specifying the size and shape of the array. (See page 3-6 for more information.) For complex numbers, pass the imaginary part to the constructor in a separate C++ array. This method of creating an array is more efficient than using concatenation.

Column-Major versus Row-Major Storage

When you are initializing a MATLAB array with a standard C++ array, store the C++ array data in column-major order. For example, to create a 2-by-3 array (two rows, three columns) containing `1 2 3` in the first row and `4 5 6` in the second, you would use a six-element, one-dimensional C++ array with the elements listed in column-major order:

```
static double data[] = { 1, 4, 2, 5, 3, 6 };
mwArray A(2, 3, data);
```

To list the data in an array in column-major order, read down the columns, from the left-most column to the right-most column. The three columns of this array are `1 4`, `2 5`, and `3 6`.

Using Row-Major Data to Create a Column-Major Array

In some cases, specifying a C++ array in column-major order is inconvenient. An additional function, `row2mat()`, creates a matrix from a C++ array that stores its data in row-major order. Rewriting the above example to use `row2mat()` yields this code:

```
static double data[] = { 1, 2, 3, 4, 5, 6 };
mwArray A = row2mat(2, 3, data);
```

The `row2mat` function takes an optional fourth argument used for creating complex arrays. The fourth argument points to a C++ array of doubles the same size as the third argument. This fourth argument contains the complex values for the `mwArray`.

Using Scalar Expansion to Fill an Array with Values

If you used the MATLAB : operator as a wildcard in an indexed assignment statement, the MATLAB scalar expansion capability fills all the elements on the same dimension as the target assignment with the value specified.

The following C++ example uses the colon wildcard in the indexing subscript. The example creates a three-dimensional array, filling the second page of the array with the value specified.

```
mwArray A;
A(colon(),colon(),2) = 5;
```

Example Program: Creating Arrays and Array I/O (ex1.cpp)

This example demonstrates how to create an array from static data. Its primary purpose is to present a simple yet complete program. The code creates two matrices, prints them, and then reads in and prints out a third matrix. Each of the numbered sections of code is explained in more detail below.

You can find the code for this example in the `<matlab>/extern/examples/cppmath` directory on UNIX systems and in the `<matlab>\extern\examples\cppmath` directory on PCs, where `<matlab>` represents the top-level directory of your installation. See “Building C++ Applications” on page 1-13 for information about building and running the example program.

```
    // ex1.cpp

    #include <stdlib.h>
(1) #include "matlab.hpp"

(2) static double data[] = { 1, 2, 3, 4, 5, 6 };

    int main(void)
    {
        // Create two matrices.
(3)     mwArray mat0(2, 3, data);
        mwArray mat1(3, 2, data);

        // Print the matrices.
(4)     cout << mat0 << endl;
        cout << mat1 << endl;

        // Read a matrix from standard in, then print the matrix to
        // standard out.
(5)     cout << "Please enter a matrix: " << endl;
        cin >> mat1;
        cout << mat1 << endl;

(6)     return (EXIT_SUCCESS);
    }
```

The numbered items in the list below correspond to the numbered sections of code example:

1 Include header files. `matlab.hpp` declares the MATLAB C++ Math Library's data types and functions. `matlab.hpp` includes `iostream.h`, which declares the input and output streams `cin` and `cout`. `stdlib.h` contains the definition of `EXIT_SUCCESS`.

2 Declare a static array of real numbers for later use as input to an `mwArray` constructor. Note that the C++ array is one-dimensional, even though it is used to create two-dimensional matrices. Because MATLAB stores its arrays in column-major order, data that initializes a MATLAB C++ Math

Library array must also be in column-major order. C++ itself, however, stores arrays in row-major order.

To arrange mwArray data in column-major order, read down the columns of an array from the leftmost column to the rightmost column. To avoid confusion, always use one-dimensional C++ arrays to initialize mwArray objects.

3 Create two matrices by using mwArray constructors, which declare and initialize the variables, mat0 and mat1. The first matrix, mat0, has two rows and three columns; the second matrix, mat1, has three rows and two columns. Each call to the constructor takes the static array data as an argument. Constructors copy data.

4 Print the matrices using the C++ standard output stream, cout. By default, objects printed with cout appear on the terminal screen, though you can redirect the output to a file. A stream is a sequence of bytes that can be read from or written to. It is more general than a file, encompassing all the I/O devices attached to a computer (keyboard, terminal screen, disk, etc.). Refer to your C++ reference for a complete explanation of streams and C++'s input and output facilities.

5 Prompt the user to type in a matrix. Read the matrix into mat1 using the C++ standard input stream, cin. The matrix does not need to be the same size as the matrix already stored in mat1. The input operator >> creates a new matrix and assigns that matrix to mat1. UNIX systems and PCs read from the terminal by default; you can redirect them to read from an input file. See "Using Array Stream I/O" in Chapter 8 to learn about the I/O format for the library and how it differs from MATLAB's.

6 Print the newly read matrix.

Output

The program prints the matrices, mat0 and mat1, and then prompts the user to enter a matrix.

```
[
        1         3         5 ;
        2         4         6
]
```

```
[
      1       4 ;
      2       5 ;
      3       6
]
```

```
Please enter a matrix:
```

If you enter a valid matrix, for example, `[ 1 0; 0 1]`, the program prints it.

```
[
      1       0 ;
      0       1
]
```

Note that the output format is the same as the input format, enabling the output from one program to be used as the input to another. The input format is simple. Matrix text begins with the character `[`. The opening bracket is followed by any number of rows of integers or floating-point numbers separated by semicolons. Each row must contain the same number of columns. Matrix text ends with a `]`. Spaces, tabs, and carriage returns are ignored. For complete information about using stream I/O, see "Using Array Stream I/O" on page 8-3.

Note Because the array input format is the same as the array output format, data written out by one program can be easily read in by another program.

Sparse Matrices

The MATLAB C++ Math Library includes routines to create and manipulate sparse matrices. Sparse matrices provides a more efficient storage format for two-dimensional numeric arrays with few non-zero elements. Only two-dimensional numeric arrays can be converted to sparse storage format.

The following table lists the MATLAB C++ Math Library routines to create sparse matrices and perform some basic tasks with them. The sections that follow provide more detail about using these routines. For more detailed information about using sparse arrays, see *Using MATLAB*. For more detailed information about any of the library routines, see the *MATLAB C++ Math Library Reference*.

Table 3-2: Sparse Matrix Routines

To ...	Use ...
Create a sparse matrix	`sparse()`
Convert a sparse matrix into a full matrix	`full()`
Replace nonzero sparse matrix elements with ones	`spones()`

Replace nonzero sparse matrix elements with random numbers.	`sprand()`
	`sprandn()`
	`sprandnsym()`

Convert a text file into a sparse matrix	`spconvert()`
Create a sparse identity matrix	`speye()`
Extract a band or diagonal group of elements from a matrix and create a sparse matrix.	`spdiags()`
Determine the number of nonzero elements in a numeric matrix.	`nnz()`

Table 3-2: Sparse Matrix Routines (Continued)

To ...	Use ...
Determine if a matrix has any nonzero elements or if all elements are nonzero.	`any()` or `all()`
Determine the amount of storage allocated for the nonzero elements of a sparse matrix.	`nzmax()`
Obtain a vector containing all the nonzero elements of a sparse matrix.	`nonzeros()`
Apply a function to all the nonzero elements of a sparse matrix	`spfun()`

Creating a Sparse Matrix

To create a sparse matrix in a C++ program, use the MATLAB C++ Math Library `sparse()` routine. Using this routine, you can create sparse arrays in two ways:

- By converting an existing array to sparse format
- By specifying the data and the location of the data in the sparse array.

Converting an Existing Matrix into Sparse Format

To create a sparse matrix from a standard numeric array, use the `sparse()` routine. `sparse()` converts the numeric array into sparse storage format. The following code fragment creates a sparse matrix from an identity matrix and then converts it to sparse format.

```
mwArray A,B;

A = eye(12);
cout << A << endl;

B = sparse(A);
cout << B << endl;
```

This code displays the identity matrix in full and sparse formats.

```
[
 1    0    0    0    0    0    0    0    0    0    0    0;
 0    1    0    0    0    0    0    0    0    0    0    0;
 0    0    1    0    0    0    0    0    0    0    0    0;
 0    0    0    1    0    0    0    0    0    0    0    0;
 0    0    0    0    1    0    0    0    0    0    0    0;
 0    0    0    0    0    1    0    0    0    0    0    0;
 0    0    0    0    0    0    1    0    0    0    0    0;
 0    0    0    0    0    0    0    1    0    0    0    0;
 0    0    0    0    0    0    0    0    1    0    0    0;
 0    0    0    0    0    0    0    0    0    1    0    0;
 0    0    0    0    0    0    0    0    0    0    1    0;
 0    0    0    0    0    0    0    0    0    0    0    1
]

ans =
   (1,1)        1
   (2,2)        1
   (3,3)        1
   (4,4)        1
   (5,5)        1
   (6,6)        1
   (7,7)        1
   (8,8)        1
   (9,9)        1
  (10,10)       1
  (11,11)       1
  (12,12)       1
```

Creating a Sparse Matrix from Data

You can also create a sparse matrix by specifying the value and location of all the nonzero elements when you create it. Using `sparse()`, you specify as arguments:

- Two vectors, `i` and `j`, that specify the row and column subscripts
- One vector, `s`, containing the real or complex data you want to store in the sparse matrix. Vectors `i`, `j` and `s` should all have the same length.
- Two scalar arrays, `m` and `n`, that specify the dimensions of the sparse matrix to be created

- An optional scalar array that specifies the maximum amount of storage that can be allocated for this sparse array

The following code example illustrates how to create a 8-by-7 sparse matrix from data. This call specifies a single value, 9, for all the nonzero elements of the sparse matrix which is replicated in all nonzero elements by scalar expansion. To see the pattern formed by this sparse matrix, see the output of this code which follows.

```
//  declare C++ arrays of index values
double inums[] = {3,4,5,4,5,6};
double jnums[] = {4,3,3,5,5,4};

// declare MATLAB arrays
mwArray S;
mwArray i(1,6,inums,NULL); // Use constructors to create
mwArray j(1,6,jnums,NULL); // initialized index vectors

// create sparse matrix
S = sparse(i, j, 9, 8, 7);

cout << S << endl;

cout << full(S) << endl;
```

This code produces the following output:

```
(4,3)   9
(5,3)   9
(3,4)   9
(6,4)   9
(4,5)   9
(5,5)   9

[
   0    0    0    0    0    0    0;
   0    0    0    0    0    0    0;
   0    0    0    9    0    0    0;
   0    0    9    0    9    0    0;
   0    0    9    0    9    0    0;
   0    0    0    9    0    0    0;
```

```
     0    0    0    0    0    0    0;
     0    0    0    0    0    0    0;
]
```

Converting a Sparse Matrix to Full Matrix Format

You can convert a sparse matrix to a full format matrix by using the `full()` routine. The previous example nested a call to this routine in `output` to show the pattern created by the values in the sparse matrix:

```
cout << full(S) << endl; // print full matrix
```

Evaluating Arrays for Sparse Storage

To see if a MATLAB array is a good candidate for sparse format storage, determine the number of nonzero elements in an array, using the `nnz()` routine. The following code fragment creates a 5-by-5 identity matrix and then obtains the number of nonzero elements in the identity matrix.

```
mwArray A = eye(5);

cout << nnz(A) << endl;
```

Character Arrays

The MATLAB C++ Math Library also includes routines to create and manipulate character arrays. One-dimensional character arrays are also called *strings*. Multidimensional character arrays are also called *arrays of strings*. In an array of strings, each string must be the same length. The routines that create arrays of strings use blanks to pad the strings to the same length. In a cell array of strings, individual strings can be different lengths. For information about cell arrays, see page 3-28.

The following table lists the MATLAB C++ Math Library routines to create character arrays and perform some basic tasks with them. The sections that follow provide more detail about using these routines. For more detailed information about using character arrays, see *Using MATLAB*. For more detailed information about any of the library routines, see the *MATLAB C++ Math Library Reference*.

Table 3-3: Character Array Routines

To ...	Use ...
Create a character array	`mwArray` string constructor: `mwArray("abcd")`
Create a character array from a numeric array	`char_func()`
Convert a character array to its underlying numeric representation.	`double_func()`

Concatenate character strings into a multidimensional, blank-padded character array	`str2mat()`
	`strcat()`
	`strvcat()`

Convert an array of blank-padded character strings into a cell array of strings	`cellstr()`
Concatenate character strings into a cell array of strings	`char_func()`

Table 3-3: Character Array Routines (Continued)

To ...	Use ...
Remove extra blank characters from individual rows in a character array.	`deblank()`
Display a character string.	`disp()`
Convert a number to its string representation, specifying format.	`num2Str()`
Round the elements in an array to integers and convert the results into a string matrix.	`int2str()`
Convert character string into a numeric array.	`str2num()`

Creating MATLAB Character Arrays

MATLAB represents characters in 16-bit, Unicode format. You can create MATLAB strings using any of the following array creation mechanisms:

- Using an `mwArray` constructor
- Converting an array of a different type into a character array
- Concatenating existing arrays

The following sections provide more detail about creating arrays with each of these mechanisms, highlighting areas where the C++ syntax is significantly different from the corresponding MATLAB syntax.

Using the Character Array Constructor

The easiest way to create a MATLAB character string is to pass a standard C++ character string to the `mwArray` string constructor.

```
mwArray A("my string");

cout << "A = " << A << endl;
```

This code produces the following output:

```
'my string'
```

Converting Numeric Arrays to Character Arrays

To convert a numeric array into a character array, use the char_func() routine. The following code creates an array of numeric values and then converts it into a string.

```
mwArray C,D;

C = horzcat(109,121,32,115,116,114,105,110,103);

D = char_func(C);

cout << D << endl;
```

This code produces the following output:

```
'my string'
```

To convert this character array back into its underlying numeric representation in double precision format, use the double_func() routine.

Creating Multidimensional Arrays of Strings

You can create a multidimensional array of MATLAB character strings; however, each string must have the same length. The MATLAB C Math Library routines that create arrays of character strings pad the strings with blanks to make them all a uniform length.

Note To create a multidimensional character array where individual strings aren't padded with blanks, use cell arrays. See page 3-28 for more information.

The following code fragment uses the char_func() routine to create a two-dimensional array of strings from two strings of different lengths.

```
mwArray Z("my string");
mwArray Y("my dog");

mwArray Q = char_func(Z,Y);

cout << Q << endl;
```

This code fragment produces the following output. Note how `char_func()` adds three blanks to the string "`my dog`" to make it the same length as "`my string`", creating a 2-by-9 character array.

```
[
    'my string';
    'my dog   ';
]
```

Cell Arrays

MATLAB cell arrays provide a way to group together a collection of dissimilar MATLAB arrays.

The following table lists the MATLAB C++ Math Library routines to create cell arrays and perform basic tasks with them. The sections that follow provide more detail about using these routines. For more detailed information about using cell arrays, see *Using MATLAB*. For more detailed information about any of the library routines, see the *MATLAB C++ Math Library Reference*.

Table 3-4: Cell Array Routines

To ...	Use ...
Create a multidimensional array of empty cells.	`cell()`
Create a cell array of strings from a character array	`cellstr()`
Create a cell array by concatenating existing arrays	`cellhcat()`
Convert a structure into a cell array	`struct2cell()`
Convert a numeric array into a cell array	`num2cell()`
View the contents of each cell in a cell array	`celldisp()`

Creating Cell Arrays

The MATLAB C++ Math Library allows you to create cell arrays by:

- Using a cell array creation function
- Using a cell array conversion function
- Concatenating existing arrays
- Assigning a value to an element in a cell array

Using a Cell Array Creation Routine

Using the MATLAB C++ Math Library `cell()` routine, you can create a cell array of any size and dimension. You specify the size of each dimension as arguments to the routine. The `cell()` routine creates an array of empty cells. The following code fragment creates a 2-by-3-by-2 cell array. If you specify a single argument, MATLAB creates a square matrix.

```
mwArray  C;

C = cell(2,3,2);

cout << C << endl;
```

This code produces the following output:

```
(:,:,1) =
    []   []   []
    []   []   []
(:,:,2) =
    []   []   []
    []   []   []
```

Using a Cell Array Conversion Routines

You can also create cell arrays by converting other MATLAB arrays into cell arrays. The MATLAB C++ Math Library includes routines that convert a numeric array into a cell array, `num2cell()`, or a structure into a cell array, `struct2cell()`.

The following code fragment creates a numeric array, using `ones()`, and converts it into a cell array using the `num2cell()` routine.

```
mwArray B = ones(2,3);

cout << "B" << B << endl;

mwArray C = num2cell(B);

cout << "C" << C << endl;
```

In this output, the brackets indicate that each element in the numeric array has been placed into a cell in the cell array.

```
B  [
    1     1     1;
    1     1     1
  ]

C   [1]  [1]  [1]
    [1]  [1]  [1]
```

The brackets indicate cell array elements.

Creating Cell Arrays through Concatenation

In MATLAB, you can create a cell array by combining groups of arrays together using the MATLAB cell concatenation operator: curly braces ({}). For example, the following MATLAB statement horizontally concatenates several individual arrays into a 1-by-4 array of cells.

```
C = { 'jon' ones(2) magic(3) 5 }

C =
    'jon'    [2x2 double]    [3x3 double]    [5]
```

To create a cell array with multiple rows in MATLAB, use the same syntax, inserting semicolons into the list of arrays at row breaks.

The MATLAB C++ Math Library uses the `cellhcat()` routine to emulate the MATLAB {} operator. For example, the following code fragment creates the same 1-by-4 cell array.

```
mwArray C;

C = cellhcat( "jon", ones(2), magic(3), 5 );

cout << “C = \n” << C << endl;
```

This code produces the following cell array:

```
C =
    'jon'    [2x2 double]    [3x3 double]    [5]
```

To create a cell array with multiple rows, you must use `cellhcat()` to create the horizontal rows and `vertcat()` to stack the rows in columns.

```
mwArray C;
```

```
C = vertcat(cellhcat("jon",ones(2)),cellhcat(magic(3), 5 ));

cout << “C = \n” << C << endl;
```

This code produces the following two-dimensional cell array:

```
C =
    'jon'           [2x2 double]
    [3x3 double]    [         5]
```

Creating Multidimensional Cell Arrays by Concatenation. To create a cell array of more than two dimensions, you must concatenate several existing cell arrays using the `cat()` routine. The cat() routine uses the following syntax

```
B = cat(dim,A1,A2...)
```

where `A1`, `A2`, and so on are the cell arrays to concatenate, and `dim` is the dimension along which to concatenate them.

For example, the following code fragment creates a pair of two-dimensional cell arrays and then uses `cat()` to concatenate them along three-dimensions to create a 2-by-2-by-2 cell array.

```
mwArray C = vertcat( cellhcat("jon",5),
                     cellhcat(magic(3),ones(2)));

mwArray D = vertcat( cellhcat("jim",zeros(3)),
                     cellhcat("joe",12));

mwArray E = cat( 3, C, D);
cout << E << endl;
```

This code produces the following output:

```
(:,:,1)
    'jon'           [         5]
    [3x3 double]    [2x2 double]

(:,:,2)
    'jim'           [3x3 double]
    'joe'           [        12]
```

Creating Cell Arrays by Assignment

When you assign a value to an element in a cell array, the MATLAB C++ Math Library creates the array (or extends an existing array) to accommodate the assignment. For cell arrays, the library supports two ways to perform this assignment:

- Cell indexing
- Content indexing

With cell indexing, you identify the target location of the assignment (the left hand side of the assignment statement) using standard MATLAB indexing syntax, and you enclose the values to be assigned to the cell array (the right hand side of the assignment statement) with the MATLAB cell concatenation ({ }) operator. For example, the following MATLAB syntax creates a 2-by-2 cell array by assignment: `D(2,2) = { 'jones' }`.

With content addressing, you use the braces ({ }) operator on the left hand side of the assignment statement to indicate that you want to affect the value contained in the cell. On the right hand side of the assignment statement, you do not need to specify braces. For example, the following MATLAB syntax creates a 2-by-2 cell array by assignment: `D{2,2} = ''jones'`.

You can use either indexing method interchangeably: the result is the same. For more information about using indexing to access elements in a cell array, see Chapter 4.

Using Cell Indexing to Create a Cell Array. The MATLAB C++ Math Library uses the `cellhcat()` routine to emulate the cell array concatenation ({ }) operator in cell indexing statements.

For example, the following code fragment creates a 2-by-2 cell array by assigning a value to element at row 2, column 2. This example is the same as the MATLAB syntax `D(2,2) = { 'jones' }`. Note that you must declare the cell array before using it in the cell indexing statement.

```
mwArray D;

D(2,2) = cellhcat("jones");

cout << D << endl;
```

This produces the following output:

```
[]       []
[]  'jones'
```

Using Content Indexing to Create a Cell Array. The MATLAB C++ Math Library uses the `mwArray::cell()` member function to emulate the use of the braces operator in content indexing statements.

The following code fragment illustrates how to use content indexing to create a cell array by assignment. Note that you must declare the array before using it in the content indexing statement. This example is the same as the MATLAB syntax `Z{2,2} = 'jones'`.

```
mwArray Z;

Z.cell(2,2) = "jones";

cout << Z << endl;
```

This produces the following output:

```
[]       []
[]  'jones'
```

Note Do not confuse the `mwArray::cell()` member function with the `cell()` library routine. The `cell()` library routine creates arrays of empty cells.

Displaying the Contents of a Cell Array

The C++ output operator (<<) is used to direct a value to an output stream such as the predefined ostream cout (standard output). For numeric arrays, the output operator displays the contents of each array element. However, for cell arrays, the output operator displays the contents of a cell only if the value stored there is a MATLAB character array or scalar array. For cells containing multidimensional arrays, cell arrays or other MATLAB arrays, the output operator only displays the size of the array stored in the cell.

To display the contents of each cell in a cell array, you must use the `celldisp()` routine. To illustrate, the following code fragment creates a cell array that contains a MATLAB character array, a scalar array, and several multidimensional arrays. The example then prints out the cell array using the output operator and celldisp().

```
mwArray C = vertcat(cellhcat("jon",5),
                    cellhcat(magic(3),
                    ones(2,3,2)));

cout << "cout output:\n" << C << endl;
cout << "celldisp() output:\n" << endl;
celldisp(C,”C”);
```

This code produces the following output:

```
cout output:
    'jon'           [         5]
    [3x3 double]    [2x2 double]

celldisp() output:

C{1,1} =
jon
C{2,1} =
     8    1    6
     3    5    7
     4    9    2

C{1,2} =
     5

C{2,2} =

(:,:,1) =
     1    1    1
     1    1    1
(:,:,2) =
     1    1    1
     1    1    1
```

MATLAB Structures

A MATLAB structure can be thought of as a one-dimensional cell array in which each cell is assigned a name. These named cells are called *fields*. You can create multidimensional arrays of structures; all the structures in an array of structures must have the same fields.

The following table lists the MATLAB C++ Math Library routines to create structures and perform basic tasks with them. The sections that follow provide more detail about using these routines. For more detailed information about using structures, see *Using MATLAB*. For more detailed information about any of the library routines, see the *MATLAB C++ Math Library Reference*.

Table 3-5: MATLAB Structure Routines

To ...	Use ...
Create a structure an initialize it with values.	`struct_func()`
Convert a cell array into a structure.	`cell2struct()`
Determine the names of the fields in a structure.	`fieldnames()`
Determine if a string is the name of a field in a structure.	`isfield()`
Access the contents of a field in a structure.	`getfield()`
Specify the value of a field in a structure.	`setfield()`
Remove a field from each structure in an array of structures.	`rmfield()`

Creating Structures

The MATLAB C++ Math Library allows you to create structures by:

- Using a structure creation routine

- Using a structure conversion routine
- Assigning a value to an element in a structure.

Using a Structure Creation Routine

You can create a structure using the `struct_func()` routine. This routine lets you define the fields in the structure and assign a value to each field. For example, the following code fragment creates a structure that contains two fields, a text string and a scalar value.

```
mwArray A;

A = struct_func("name",   // field name
                "John",   // value
                "number", // field name
                 311);    // value

cout << A << endl;
```

This code produces the following output:

```
  name: 'John'
number: 311
```

Note The `struct_func()` routine defines the fields and their values in a single instance of a structure. To create an array of structures, use MATLAB indexing syntax in the assignment statement, as described in “Using Assignment to Create Structures” on page 3-37.

Using a Structure Conversion Routine

You can also create structures by converting an existing MATLAB cell array into a structure, using the `cell2struct()` routine. When converting a cell array into a structure, you must create a separate cell array that contains the names you want to assign to fields in the structure.

The following code fragment creates a cell array, `C`, containing data and a second cell array, `F`, containing field names. The example then passes these cell arrays to `cell2Struct()`.

```
mwArray C,F,S;

// create cell array to be converted

C = cellhcat("tree",
               37.4,
             "birch");

cout << C << endl;

// create cell array of field names

F = cellhcat("category",
             "height",
             "name");

// convert cell array to structure

S = cell2struct(C,F,2);

cout << S << endl;
```

This code generates the following output:

```
'tree'   [37.4000]    'birch'

category: 'tree'
  height: 37.4000
    name: 'birch'
```

Using Assignment to Create Structures

As with other MATLAB arrays, if you assign values to fields in a structure that is in an array of structures, the MATLAB C++ Math Library creates an array of structures large enough to accommodate the location specified by the index string.

The following example defines a structure with two fields, `name` and `number`. Because it is an indexed assignment statement, the library creates an array of 3 of these structures, assigning values to the third structure in the array. The first two structures in the array are initialized to contain empty arrays for each field. This C++ code is equivalent to the MATLAB statement, `S(2) = struct('name','jim','number',312)`.

```
mwArray S;

// Create array of structures by assignment

S(3) = struct_func("Name",    // Field name
                   "Jim",     // Value
                   "Number",  // Field name
                    312);     // Value

cout << S << endl;
```

This code generates the following output:

```
1x3 struct array with fields
    Name
    Number
```

For more detailed information about using indexing, see Chapter 4.

Performing Common Array Programming Tasks

The following sections describe some common programming tasks that you must perform for all types of MATLAB array.

Converting Data to MATLAB Arrays

The operators and functions in the MATLAB C++ Math Library operate on arrays and produce arrays as results. However, because all data is not available in array form, the library provides functions for converting data to and from arrays. In general, anywhere the library interface requires an `mwArray`, you can use a data type that can be converted to an `mwArray`.

In C++, two types of routines, constructors and casts, handle the conversions. Constructors transform raw data into C++ objects. Casts extract data from already-constructed objects. Constructors always result in new objects, whereas casts either produce new objects or provide pointers to the data in the original objects. C++ automatically performs many of these conversions for you.

Converting Data into a MATLAB Array

You can convert five types of data to an `mwArray`:

- A scalar
- A string
- An array of double-precision floating-point numbers
- A MATLAB `mxArray` pointer, also known as a `MatlabMatrix` pointer
- An `mwSubArray`

A new `mwArray` object is created when any of these data types is converted to an `mwArray` object.

The most common conversions are from scalars, strings, and arrays of doubles to `mwArrays`. If you are working with MEX-Files or the MATLAB C Math Library, you may need to convert the `mxArray` pointers that those routines return into `mwArray` objects since the MATLAB C++ Math Library does not handle `mxArray` pointers. `mwSubArray` objects result from indexing operations; the library itself handles them for you.

The table below demonstrates how to convert the various data types to an mwArray. The table shows an implicit conversion and an explicit conversion for each data type. C++ automatically performs implicit conversions for you. Explicit conversions are ones that you can explicitly invoke.

For most uses, the code in the implicit column is sufficient. C++ determines which constructor to invoke from the types of the operands on either side of the assignment statement. In some cases, however, C++ may not be able to determine unambiguously which conversion to apply, and an explicit conversion may be necessary.

Table 3-6: Converting to an mwArray

From...	Implicit Constructor	Explicit Constructor
Scalar	`mwArray A;` `A = 5;`	`mwArray A;` `A = mwArray(5)`
String	`mwArray A;` `A = "abcd";`	`mwArray A;` `A = mwArray("abcd");`

Array of doubles	Not Available	`mwArray A;`
		`static double x[]={1,5};`
		`A = mwArray(1,2,x);`

mxArray pointer	`mwArray A;`
	`mxArray *mat;`
	`A = mat;`
		`mwArray A;`
		`mxArray *mat;`
		`A = mwArray(mat);`

mwSubArray	`mwArray A, B;`
	`B = ramp(1,10);`
	`mwSubArray sub=B(8);`
	`A = sub;`
		`mwArray A, B;`
		`B = ramp(1,10);`
		`mwSubArray sub = B(8);`
		`A = mwArray(sub);`

You can also use cast syntax in the explicit case. See a C++ manual for more information about the equivalence between constructors and casts.

Converting a MATLAB Array into Data

mwArrays can be converted into two types of data:

- Scalars
- MATLAB mxArray pointers

The table below demonstrates how to extract data from an mwArray. In the pointer case (mxArray pointer), the conversion returns a pointer to the internal data of the mwArray object rather than to a new object. Take care not to modify the data referenced by the pointer. The returned pointer is defined as const to remind you that the data it points to should not be modified.

There are two limitations to the types of mwArray you can cast into a double:

- You cannot assign a nonscalar mwArray to a C++ scalar variable (int or double). You can only cast 1-by-1 arrays to scalars.
- You cannot cast a complex mwArray (scalar or nonscalar) to a double-precision scalar or an array. Before assigning a complex array to a real-valued variable, convert the complex mwArray to a real mwArray with the real() or imag() functions.

Both of these cases raise an exception.

Table 3-7: Extracting Data from an mwArray

To...	Implicit Cast	Explicit Cast

Integer	`int32 i;`
	`mwArray A = 5;`
	`i = A;`
		`int32 i;`
		`mwArray A = 5;`
		`i = (int32)A;`

Double	`double x;`
	`mwArray A = 5;`
	`x = A;`
		`double x;`
		`mwArray A = 5;`
		`x = (double)A;`

Refer to “Extracting Data from an mwArray” in Chapter 10 to learn about mwArray::GetData() and mwArray::ToString().

Efficiency Considerations

Conversions are not always efficient operations. It is important to minimize their use in certain situations. In particular, using a scalar array as a loop index bound is very inefficient. You obtain much better performance by first converting the mwArray to an integer and then using the integer as the loop bound variable.

The following code demonstrates an inefficient use of an mwArray as a loop bound variable. In each iteration of the for-loop, the comparison i < A requires that A be converted from an mwArray to a scalar. This conversion is expensive.

```
// Inefficient loop bound variable
mwArray A = 5;
int i;
for (i=0; i<A; i++)
    cout << "Counting: " << i << endl;
```

The code below runs much faster because the variable A is explicitly cast to an integer before the loop begins; integer j rather than A is used as the for-loop bound variable.

```
// Efficient loop bound variable
mwArray A = 5;
int i, j = A;
for (i=0; i < j; i++)
     cout << "Counting: " << i << endl;
```

In this case, the cast operation is invoked only once.

Determining Array Size

You can determine the size of an array in several ways:

- Using the size() routine, which returns the number of rows and columns in an array
- Using an overloaded version of size() that returns integers
- Using the mwArray::Size() member function
- Using the mwArray::EltCount() member function

Using the size() Routine

```
mwArray A = rand(4,7) ;
mwArray m, n;
size(mwVarargout(m,n), A);
```

This version of size() returns the number of rows and columns in one or more separate arrays. Because it returns the dimensions as MATLAB scalar arrays, size() is consistent with the rest of the interface. However, because it returns arrays, it is far slower and uses more storage than the overloaded version of size() that returns dimensions as integers.

Using the Overloaded size() Routine

```
mwArray A = rand(4,7);
int m, n;
m = size(&n, A);
```

The overloaded version of size() returns array dimensions as integers rather than scalar arrays. This version of size() is efficient and easy to use; however, it only supports two dimensional arrays. It has been superseded by the member function, mwArray::Size(), which works for multidimensional arrays.

Using the mwArray Size() Member Functions

The mwArray object supports several member functions that returns array dimensions as integers rather than scalar arrays. These member functions are the most efficient way to compute the dimensions of an array.

The following examples show the various ways to determine array dimensions using these member functions.

```
int32 dims[2];
int32 ndims[3];

mwArray A = rand(4,7);      // Two-dimensional array
mwArray B = rand(4,7,4);    // Three-dimensional array

A.Size(dims);       // Sets dims to (4, 7), maxdims defaults to 2
B.Size(ndims,3);     // Sets ndims to (4, 7, 4)
A.Size();            // Returns the number of dimensions: 2
A.Size(1);           // Returns the size of the 1st dimension: 4
A.Size(2);           // Returns the size of the 2nd dimension: 7
```

mwArray EltCount() Member Function

In addition to the size member functions, the mwArray object includes a member function, named EltCount(), that returns the number of elements in the array, determined by the product of the length of each dimension:

```
A.EltCount();        // Returns the product of M and N: 28
```

Note The capitalization of these function names is significant; C++ function names are case-sensitive.

4

Indexing into Arrays

Overview

The MATLAB interpreter provides a sophisticated and powerful indexing operator that accesses and modifies multiple array elements. The MATLAB C++ Math Library also supports an indexing operator. This chapter describes how to:

- Use one-dimensional, n-dimensional, and logical subscripts
- Make assignments using indexing expressions
- Make deletions using indexing
- Index into cell arrays
- Index into structure arrays

Terminology

This diagram illustrates the terminology used in this chapter.

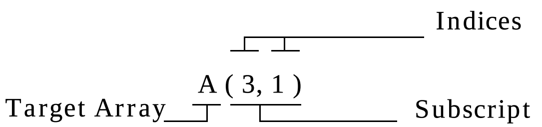


Figure 4-1: From the MATLAB and MATLAB C++ Math Library Perspective

Dimensions and Subscripts in MATLAB

There are three types of data in MATLAB: numeric arrays, cell arrays, and structures (objects are just a special kind of structure). Therefore, there are three types of indexing, one for each type of data:

- Standard indexing, which uses parentheses `()`
- Cell array indexing, which uses curly braces `{}`
- Structure indexing, which uses named fields, for example, `color`

Both standard indexing and cell array indexing take numeric arguments, one argument for each dimension of the array being indexed into, while structure indexing uses only the name of the structure field.

Note Standard indexing can be used with all three types of data, while cell array indexing can only be used on cell arrays, and structure indexing only on structures. You can combine, for example, standard indexing and structure indexing on a structure.

Dimensions and Subscripts in the MATLAB C++ Math Library

The MATLAB C++ Math Library supports N-dimensional standard, cell array, and structure indexing. You use:

- The indexing operator `()` for standard indexing
- `mwArray::cell()` for cell array indexing
- `mwArray::field()` for structure indexing

The indexing operator `()` and `mwArray::cell()` take numeric arguments, one index for each dimension of the array being indexed. `mwArray::field()` takes the name of the structure field as an argument.

Note You cannot index into an array with more dimensions than the array has, although you can use fewer dimensions.

Applying a subscript to an array allows you to:

- Access
- Modify
- Delete

elements of an array. For example, the two-dimensional indexing expression

```
A(3,1)
```

applies the subscript `(3,1)` to `A` and returns the element at row 3, column 1. `A(9)`, a one-dimensional indexing expression, returns the ninth element of array `A`.

Note The indexing functions follow the MATLAB convention for array indices: indices begin at one rather than zero.

An index can be a scalar, a vector, a matrix, or a call to the special function `colon()`.

- A scalar subscript selects a scalar value.
- A subscript with vector or matrix indices selects a vector or matrix of values.
- The `colon()` index, which loosely interpreted means "all," selects, for example, all the columns in a row or all the rows in a column.

If you provide arguments to `colon()`, the subscript specifies a vector. For example, `colon(1,10)` specifies the vector `[ 1 2 3 4 5 6 7 8 9 10 ]`.

Tip `for`-loops provide an easy model for thinking about indexing. A one-dimensional index is equivalent to a single `for`-loop; a two-dimensional index is equivalent to two nested `for`-loops. The size of the subscript determines the number of iterations of the `for`-loop. The value of the subscript determines the values of the loop iteration variables.

The MATLAB C++ Math Library implements indexing via the interaction of three classes: `mwArray`, `mwIndex`, and `mwSubArray`. `mwArray` represents the array itself. `mwIndex` represents an index. `mwSubArray` represents the result of an index operation. The indexing routines themselves create `mwSubArray` objects when an indexing expression appears as the target of an assignment operation (on the left-hand side of an assignment operator). The library handles `mwSubArray` objects for you; you do not need to create them.

Array Storage

MATLAB stores each array as a column of values regardless of the actual dimensions. This column consists of the array columns, appended top to bottom. For example, MATLAB stores

```
A = [2 6 9; 4 2 8; 3 0 1]
```

as

```
2
4
3
6
2
0
9
8
1
```

Accessing A with a single subscript indexes directly into the storage column. A(3) accesses the third value in the column, the number 3. A(7) accesses the seventh value, 9, and so on.

If you supply more subscripts, MATLAB calculates an index into the storage column based on the dimensions you assigned to the array. For example, assume a two-dimensional array like A has size [d1 d2], where d1 is the number of rows in the array and d2 is the number of columns. If you supply two subscripts (i,j) representing row-column indices, the offset is

```
(j–1)*d1+i
```

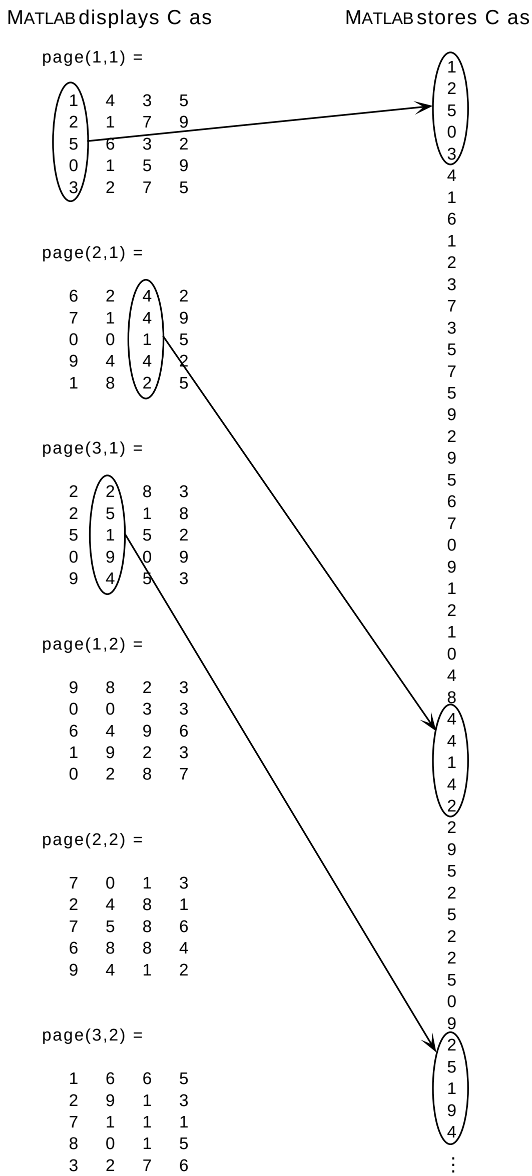
Given the expression A(3,2), MATLAB calculates the offset into A's storage column as (2-1)*3+3, or 6. Counting down six elements in the column accesses the value 0.

This storage and indexing scheme also extends to multidimensional arrays. You can think of an N-dimensional array as a series of "pages," each of which is a two-dimensional array. The first two dimensions in the N-dimensional array determine the shape of the pages, and the remaining dimensions determine the number of pages.

In a three- (or higher) dimensional array, for example, MATLAB iterates over the pages to create the storage column, again appending elements columnwise. You can think of three-dimensional arrays as "books," with a two-dimensional array on each page. The term *page* is used frequently in this document to refer to a two-dimensional array that is part of a larger N-dimensional array.

Labeling the dimensions past three is more difficult. You can imagine shelves of books for dimension 4, rooms of shelves for dimension 5, libraries of rooms for dimension 6, but at that point the analogy loses meaning. This document rarely uses an array of dimension greater than three or four, although MATLAB and the MATLAB C Math Library handle any number of dimensions that doesn't exceed the amount of memory available on your computer.

For example, consider a 5-by-4-by-3-by-2 array `C`.


MATLAB displays C as
MATLAB stores C as
page(1,1) =
1 4 3 5
2 1 7 9
5 6 3 2
0 1 5 9
3 2 7 5
page(2,1) =
6 2 4 2
7 1 4 9
0 0 1 5
9 4 4 2
1 8 2 5
page(3,1) =
2 2 8 3
2 5 1 8
5 1 5 2
0 9 0 9
9 4 5 3
page(1,2) =
9 8 2 3
0 0 3 3
6 4 9 6
1 9 2 3
0 2 8 7
page(2,2) =
7 0 1 3
2 4 8 1
7 5 8 6
6 8 8 4
9 4 1 2
page(3,2) =
1 6 6 5
2 9 1 3
7 1 1 1
8 0 1 5
3 2 7 6
1
2
5
0
3
4
1
6
1
2
3
7
3
5
7
5
9
2
9
5
6
7
0
9
1
2
1
0
4
8
4
4
1
4
2
2
9
5
2
5
2
2
5
0
9
2
5
1
9
4
⋮

Again, a single subscript indexes directly into this column. For example, `C(4)` produces the result

```
ans =

     0
```

If you specify two subscripts `(i,j)` indicating row-column indices, MATLAB calculates the offset as described above. Two subscripts always access the first page of a multidimensional array, provided they are within the range of the original array dimensions.

If more than one subscript is present, all subscripts must conform to the original array dimensions. For example, `C(6,2)` is invalid, because all pages of `C` have only five rows.

If you specify more than two subscripts, MATLAB extends its indexing scheme accordingly. For example, consider four subscripts `(i,j,k,l)` into a four-dimensional array with size `[d1 d2 d3 d4]`. MATLAB calculates the offset into the storage column by

```
(l–1)(d3)(d2)(d1)+(k–1)(d2)(d1)+(j–1)(d1)+i
```

For example, if you index the array `C` using subscripts `(3,4,2,1)`, MATLAB returns the value 5 (index 38 in the storage column).

In general, the offset formula for an array with dimensions $[d_1\ d_2\ d_3\ \ldots\ d_n]$ using any subscripts $(s_1\ s_2\ s_3\ \ldots\ s_n)$ is:

$$(s_n-1)(d_{n-1})(d_{n-2})\ldots(d_1)+(s_{n-1}-1)(d_{n-2})\ldots(d_1)+\ldots+(s_2-1)(d_1)+s_1$$

Because of this scheme, you can index an array using any number of subscripts. You can append any number of 1s to the subscript list because these terms become zero. For example, `C(3,2,1,1,1,1,1,1)` is equivalent to `C(3,2)`

Using One-Dimensional Subscripts

This section describes how to select:

- A single element with a one-dimensional scalar index
- A vector with a one-dimensional vector index
- A subarray with a one-dimensional matrix index
- All elements in a matrix with a `colon` index

All examples work with example matrix `A`. Notice that the value of each element in `A` is equal to that element's position in the column-major enumeration order. For example, the third element of `A` is the number 3 and the ninth element of `A` is the number 9.

```
A =
1 4 7
2 5 8
3 6 9
```

Overview

A one-dimensional subscript contains a single index, which can be a scalar, a vector, a matrix, or a call to the `colon()` function. The size and shape of the one-dimensional index determine the size and shape of the result. For example, a one-dimensional column vector index produces a one-dimensional column vector result.

To apply a one-dimensional subscript to an N-dimensional array, you need to know how to go from the one-dimensional index value to a location inside the array. See "Array Storage" on page 4-4 for complete details on how MATLAB counts one-dimensionally through arrays of N dimensions.

Note The range for a one-dimensional index depends on the size of the array. For a given array `A`, it ranges from 1, the first element of the array, to `prod(size(A))`, the last element in an N-dimensional array. Contrast this range with the two ranges for a two-dimensional index where the row value varies from 1 to M, and the column value from 1 to N.

Selecting a Single Element

Use a scalar index to select a single element from an array. For example,

```
A(5)
```

selects the fifth element of A, the number 5.

Selecting a Vector

Use a vector index to select multiple elements from an array. For example,

```
A(horzcat(2,5,8))
```

selects the second, fifth and eighth elements of the matrix A:

```
2 5 8
```

Because the index is a 1-by-3 row vector, the result is also a 1-by-3 row vector.

The expression

```
A(vertcat(2,5,8))
```

selects the same elements of A, but returns the result as a column vector because `vertcat()` produced a column vector:

```
2
5
8
```

Specifying a Vector Index with end( )

Sometimes you don't know how large an array is in a particular dimension, but you want to perform an indexing operation that requires you to specify the last element in that dimension. In MATLAB, you can use the end function to refer to the last element in a given dimension.

For example, `A(6:end)` selects the elements from `A(6)` to the end of the array. The MATLAB C++ Math Library's `end()` function corresponds to the MATLAB `end()` function. Given an array, a dimension (1 = row, 2 = column, 3 = page, and so on), and the number of indices in the subscript, `end()` returns (as a 1-by-1 array) the index of the last element in the specified dimension. You can then use that scalar array to generate a vector index.

Given the row dimension for a vector or scalar array, `end()` returns the number of columns. Given the column dimension for a vector or scalar array, it returns the number of rows. For a matrix, `end()` treats the matrix as a vector and returns the number of elements in the matrix.

This C++ code selects all but the first five elements in matrix `A`, just as `A(6:end)` does in MATLAB.

```
A(colon(6, end(A, 1, 1)))
```

The second argument (`1`) to `end()` identifies the dimension where `end()` is used, here the row dimension. The third argument (`1`) indicates the number of indices in the subscript; for one-dimensional indexing, it is always one. This code selects these elements from matrix `A`:

```
6 7 8 9
```

Selecting a Matrix

Use a matrix index to select a matrix. A matrix index works just like a vector index, except the result is a matrix rather than a vector. For example, let `B` be the index matrix:

```
1 2
3 2
```

Then, `A(B)` is:

```
1 2
3 2
```

Note that the example matrix `A` was chosen so that `A(X) = X` for all types of one-dimensional indexing. This is not generally the case. For example, if `A` were changed to `A = magic(3)`,

```
8 1 6
3 5 7
4 9 2
```

then `A(B)` would equal

```
8 3
4 3
```

Note In both cases, `size(A(B))` is equal to `size(B)`. This is a fundamental property of one-dimensional indexing.

Selecting the Entire Matrix As a Column Vector

Use the `colon()` index to select all the elements in a matrix. The result is a column vector. For example, `A(colon())` is:

```
1
2
3
4
5
6
7
8
9
```

The `colon()` index means "all." Think of it as a context-sensitive function. It expands to a vector array containing all the indices of the dimension in which it is used (its context). In the context of an M-by-N matrix `A`, `A(colon())` is equivalent to `A(transpose(ramp(1,M*N)))`.

Using N-Dimensional Subscripts

This section describes how to:

- Extract a scalar from a matrix
- Extract a vector from a matrix
- Extract a subarray from a matrix
- Extend two-dimensional indexing to N-dimensions

There is no functional difference between two-dimensional indexing and N-dimensional indexing (where N > 2). Because it is easier to understand two-dimensional arrays, most of the examples in this section deal with two-dimensional arrays. See “Extending Two-Dimensional Indexing to N Dimensions” on page 4-17 to learn how to work with arrays of dimension greater than two.

All two-dimensional examples work with example matrix `A`.

```
1 4 7
2 5 8
3 6 9
```

Overview

An N-dimensional subscript contains N indices. The first index is the row index, the second is the column index, the third the page index, and so on. Each index can store a scalar, vector, matrix, or the result from a call to the function `colon()`.

The size of the indices rather than the size of the subscripted matrix determines the size of the result; the size of the result is equal to the product of the sizes of the N indices. For example, assume matrix `A` is set to:

```
1 4 7
2 5 8
3 6 9
```

If you index matrix `A` with a 1-by-5 vector and a scalar, the result is a five-element vector: five elements in the first index times one element in the second index. If you index matrix `A` with a three-element row index and a two-element column index, the result has six elements arranged in three rows and two columns.

Selecting a Single Element

Use two scalar indices to extract a single element from a matrix.

For example,

```
A(2,2)
```

selects the element `5` from the center of matrix `A` (the element at row 2, column 2).

Selecting a Vector of Elements

Use a scalar index with either a vector or a matrix index to extract a vector of elements from a matrix. You can use the function `horzcat()`, `vertcat()`, or `colon()` to make the vector or matrix index, or use an `mwArray` variable that contains a vector or matrix.

The indexing routines iterate over the vector index, pairing each element of the vector with the scalar index. Think of this process as applying a (scalar, scalar) subscript multiple times; the result of each selection is collected into a vector. The indexing code iterates down the columns of the matrix index in exactly the same way it iterates over a vector index.

For example, `A(horzcat(1,3), 2)` selects the first and third element (or first and third rows) of column 2:

```
4
6
```

In MATLAB `A([1 3], 2)` performs the same operation.

If you reverse the positions of the indices, `A(2, horzcat(1, 3))`, you select the first and third element (or first and third columns) of row 2:

```
2 8
```

If the vector index repeats a number, the same element is extracted multiple times. For example, `A(2, horzcat(3, 3))` returns two copies of the element at `A(2,3)`:

```
8 8
```

Specifying a Vector Index with end( )

The end( ) function, which corresponds to the MATLAB end( ) function, provides another way of specifying a vector index. Given an array, a dimension (1 = row, 2 = column, 3 = page, and so on), and the number of indices in the subscript, end( ) returns the index of the last element in the specified dimension. You then use that scalar array to generate a vector index. See "Specifying a Vector Index with end( )" on page 4-10 for a more complete description of how and why you use the end function in MATLAB.

Given the row dimension, end( ) returns the number of columns. Given the column dimension, it returns the number of rows.

This code selects all but the first element in row 3:

```
A(3, colon(2, end(A, 2, 2)));
```

just as

```
A(3, 2:end)
```

does in MATLAB.

The second argument end( ), 2, identifies the dimension where end( ) is used, here the column dimension. The third argument, 2, indicates the number of indices in the subscript; for two-dimensional indexing, it is always 2. This code selects these elements from matrix A:

```
6   9
```

Selecting a Row or Column

Use the colon( ) index and a scalar index to select an entire row or column. For example, A(1, colon( )) selects the first row:

```
1 4 7
```

A(colon( ), 2) selects the second column:

```
4
5
6
```

Selecting a Matrix

Use two vector indices or a vector index and a matrix index to extract a matrix. You can use the function `horzcat()`, `vertcat()`, or `colon()` to make the vector or matrix index, or use `mwArray` variables that contain vectors or matrices.

The indexing code iterates over two vector indices in a pattern similar to a doubly nested `for`-loop:

```
for each element I in the row index
    for each element J in the column index
        select the matrix element A(I,J)
```

For each of the indicated rows, this operation selects the column elements at the specified column positions. For example,

```
A(horzcat(1,2), horzcat(1,3,2))
```

selects the first, third, and second (in that order) elements from rows 1 and 2, yielding:

```
1 7 4
2 8 5
```

Notice that the result has two rows and three columns. The size of the result matrix matches the size of the index vectors: the row index had two elements, the column index had three elements, so the result is 2-by-3.

The two-dimensional indexing routines treat a matrix index as one long vector, moving down the columns of the matrix. The loop for a subscript composed of a matrix in the row position and a vector in the column position works like this:

```
for each column I in the row index matrix B
    for each row J in the Ith column of B
        for each element K in the column index vector
            select the matrix element A(B(I,J), K)
```

For example, let the matrix `B` equal:

```
1 1
2 3
```

Then the expression `A(B, horzcat(1, 2))` selects the first, second, first, and third elements of columns 1 and 2:

```
1 4
```

```
2 5
1 4
3 6
```

Note that the result has two columns because `horzcat(1, 2)` has two columns.

Selecting Entire Rows and Columns

Use the `colon()` index and a vector or matrix index to select multiple rows or columns from a matrix. For example,
`A(horzcat(2, 3), colon())` selects all the elements in rows 2 and 3:

```
2 5 8
3 6 9
```

You can use `colon()` in the row position as well. For example, the expression `A(colon(), horzcat(3, 1))` selects all the elements in columns 3 and 1, in that order:

```
7 1
8 2
9 3
```

Subscripts of this form make duplicating the rows or columns of a matrix easy. See the "Duplicating a Row or Column" on page 4-44 to learn another technique for duplicating rows and columns.

Selecting an Entire Matrix

Using the `colon()` index as both the row and column index selects the entire matrix. Although this usage is valid, referring to the matrix itself without subscripting is much easier.

Extending Two-Dimensional Indexing to N Dimensions

Two-dimensional indexing extends very naturally to N-dimensions; simply use more indices. Let `A` be a 3-by-3-by-2 three-dimensional array (two 3-by-3 pages):

Page 1:

```
1 4 7
2 5 8
```

```
3 6 9
```

Page 2:

```
10 13 16
11 14 17
12 15 18
```

Then the MATLAB expression `A(:,:,2)` selects all of page 2; `A(1,:,:)` selects all the columns in row 1 on all the pages; `A(2,2,2)` selects the element at the middle of page 2 (the number `14`), and so on.

It is very simple to convert these MATLAB indexing expressions into MATLAB C++ Math Library indexing expressions.

`A(:,:,2)` becomes

```
A(colon(),colon(),2)
```

The result of this operation is the 3-by-3 array on page 2 of `A`:

```
10 13 16
11 14 17
12 15 18
```

`A(1,:,:)` becomes

```
A(1,colon(),colon())
```

The result of this operation is a three-dimensional array 1-by-3-by-2 in which each “page” consists of the first row of the corresponding page of `A`.

Page 1:

```
1 4 7
```

Page 2:

```
10 13 16
```

Finally, `A(2,2,2)` becomes:

```
A(2,2,2)
```

The result of this operation is the 1-by-1 array `14`.

If the array A had more than three dimensions, you would use more than three indices. All of the other types of indexing discussed in this chapter (selecting entire rows and columns, etc.) work equally well on N-dimensional arrays.

Using Logical Subscripts

This section describes how to use:

- A logical index as a one-dimensional subscript
- Two logical vectors as indices in a two-dimensional subscript
- A colon index and a logical vector as a two-dimensional subscript
- A logical index to select elements from a row or column

The examples work with matrix `A` and the logical array `B`.

```
A
1 4 7
2 5 8
3 6 9

B
1 0 1
0 1 0
1 0 1
```

Overview

Logical indexing is a special case of *n*-dimensional indexing. A logical index is a vector or a matrix that consists entirely of ones and zeros. Applying a logical subscript to a matrix selects the elements of the matrix that correspond to the nonzero elements in the subscript.

Logical indices are generated by the relational operators (<, >, <=, >=, ==, `!=`) and by the function `logical()`. Because the MATLAB C++ Math Library attaches a logical flag to a logical matrix, you cannot create a logical index simply by assigning ones and zeros to a vector or matrix.

You can form an *n*-dimensional logical subscript by combining a logical index with scalar, vector, matrix, or `colon()` indices.

Using a Logical Matrix As a One-Dimensional Index

When you use a logical matrix as an index, the result is a column vector. For example, you can create the logical index matrix `B`:

```
1 0 1
```

```
0 1 0
1 0 1
```

by calling `mlfLogical()`.

```
mwArray B = logical(vertcat(horzcat(1,0,1),
                            horzcat(0,1,0),
                            horzcat(1,0,1)));
```

Then `A(B)` equals:

```
1
3
5
7
9
```

Notice that `B` has ones at the corners and in the center, and that the result is a column vector of the corner and center elements of `A`.

If the logical index is not the same size as the subscripted array, the logical index is treated like a vector. For example, if `B = [1 0; 0 1]` then `A(B)` equals

```
1
4
```

since `B` has a zero at positions 2 and 3, and a 1 at positions 1 and 4. Logical indices behave just like regular indices in this regard.

Using Two Logical Vectors as Indices

Two vectors can be logical indices into an M-by-N matrix `A`. The size of a logical vector index often matches the size of the dimension it indexes though this is not a requirement.

For example, let `B = [1 0 1]` and `C = [0 1 0]`, two 1-by-3 logical vectors. Then, `A(B, C)` is

```
4
6
```

`B`, the row index vector, has nonzero entries in the first and third elements. This selects the first and third rows. `C`, the column index vector, has only one nonzero entry, the second element. This selects the second column. The result is the intersection of the two sets selected by `B` and `C`, all the elements in the second columns of rows 1 and 3.

Or, if `B = [1 0]` and `C = [0 1]`, then `A(B,C)` equals:

```
4
```

This is tricky. `B`, the row index, selects row 1. `C`, the column index, selects column 2. There is only one element in array `A` in both row 1 and column 2, the element 4.

Using One colon() Index and One Logical Vector as Indices

This type of indexing is very similar to the two-vector case. Here, however, the `colon()` index selects all of the elements in a row or column, acting like a vector of ones the same size as the dimension to which it is applied. The logical index works just like a nonlogical index in terms of size.

For example, let the index vector `B = [1 0 1]`. Then `A(colon(), B)` equals:

```
1 7
2 8
3 9
```

The `colon()` index selects all rows and `B` selects the first and third columns in each row. The result is the intersection of these two sets, that is, the first and third columns of the matrix.

For comparison, `A(B,colon())` equals:

```
1 4 7
3 6 9
```

`B` selects the first and third rows, and `colon()` selects all the columns in each row. The result is the intersection of the sets selected by each index: the first and third rows of the matrix.

Using a Scalar and a Logical Vector

Let matrix `X` be a 4-by-4 magic square.

```
X = magic(4);
```

```
16   2   3  13
 5  11  10   8
 9   7   6  12
 4  14  15   1
```

Let `B` be a logical matrix that indicates which elements in row 2 of matrix `X` are greater than 9. `B` is the result of the greater than (>) operation

```
B = X(2, colon()) > 9;
```

and contains the vector

```
0 1 1 0
```

Use `B` as a logical index that selects those elements from matrix `X`.

```
X(2,B)
```

selects these elements:

```
11 10
```

Extending Logical Indexing to N Dimensions

Logical indexing works on *n*-dimensional arrays just as you'd expect. The logical filtering happens the same way, and the subscript size governs the result size in the same manner. For details on the syntax, see "Extending Two-Dimensional Indexing to N Dimensions" on page 4-17.

Using Indexing in Assignment Statements

This section describes how to assign:

- A single element to an array
- Multiple elements to an array
- Values to all the elements in an array

The examples work with matrix A.

```
A =

   1  4  7
   2  5  8
   3  6  9
```

There is no functional difference between two-dimensional indexed assignment and *n*-dimensional indexed assignment (where N > 2). Because it is easier to understand two-dimensional arrays, most of the examples in this section deal with two-dimensional arrays. See "Extending Two-Dimensional Assignment to N Dimensions" on page 4-27 to learn how to work with arrays of dimension greater than two.

Overview

You can use any indexing expression – an array together with one or more subscripts – as the target of an assignment statement. An assignment statement consists of a destination to the left of the equals (=) operator and a source to the right. When the destination is an indexing expression, the indexing expression selects the elements that are to be modified; the source specifies the new values for those elements.

You can use five different kinds of indices:

- Scalar
- Vector
- Matrix
- Colon
- Logical

The examples below do not present all the possible combinations of these index types.

Note The size of the destination array (after the subscript has been applied) and the size of the source array must be the same.

Assigning to a Single Element

Use one or two scalar indices to assign a value to a single element in a matrix. For example,

```
A(2,1) = 17
```

changes the element at row 2 and column 1 to the integer 17. Here, both the source and destination (after the subscript has been applied) are scalars, and thus the same size.

Assigning to a Multiple Elements

Use a vector index to modify multiple elements in a matrix.

The `colon()` index frequently appears in the subscript of the destination because it allows you to modify an entire row or column. For example, the code

```
A(2,colon()) = ramp(1,3);
```

replaces the second row of an M-by-3 matrix with the vector `1 2 3`. If we use the example matrix `A`, `A` is modified to contain:

```
1 4 7
1 2 3
3 6 9
```

You can also use a logical index to select multiple elements. For example, the assignment statement

```
A(A>5)= horzcat(17,17,17,17);
```

changes all the elements in `A` that are greater than `5` to `17`:

```
1   4  17
2   5  17
```

```
3  17  17
```

Assigning to a Subarray

Use two vector indices to generate a matrix destination. For example, let the vector index B equal 1 2, and the vector index C equal 2 3. Then,

```
A(B,C) = vertcat(horzcat(1, 4) horzcat(3, 2));
```

copies a 2-by-2 matrix into the second and third columns of rows 1 and 2: the upper right corner of A. The example matrix A becomes:

```
1 1 4
2 3 2
3 6 9
```

You can also use a logical matrix as an index. For example, let B be the logical matrix:

```
0 1 1
0 1 1
0 0 0
```

Then,

```
A(B) = vertcat(horzcat(1, 4) horzcat(3, 2));
```

changes A to:

```
1 1 4
2 3 2
3 6 9
```

Assigning to All Elements

Use the colon() index to replace all the elements in a matrix with alternate values. The colon() index, however, is infrequently used in this context because you can accomplish approximately the same result by using an assignment without any indexing. For example, although you can write:

```
A(colon()) = rand(3);
```

writing

```
A = rand(3);
```

is simpler.

The first statement reuses the storage already allocated for A. The first statement will be slightly slower because the elements from the source must be copied into the destination.

Note `rand(3)` is equivalent to `rand(3,3)`.

Extending Two-Dimensional Assignment to N Dimensions

Two-dimensional assignment extends naturally to N-dimensions; simply use more indices. Let A be a 3-by-3-by-2 three-dimensional array (two 3-by-3 pages):

Page 1:

```
1 4 7
2 5 8
3 6 9
```

Page 2:

```
10 13 16
11 14 17
12 15 18
```

Then the MATLAB expression `A(:,:,2) = eye(3)` changes page 2 to the 3-by-3 identity matrix; `A(1,:,:) = ones(1, 3, 2)` changes row 1 on both pages to be all ones; `A(2,2,2) = 42` changes the element at the middle of page 2 (the number 14) to the number 42, and so on.

It is very simple to convert these MATLAB indexed assignment expressions into MATLAB C++ Math Library indexed assignment expressions.

`A(:,:,2) = eye(3)` becomes

```
A(colon(),colon(),2) = eye(3);
```

As a result of this operation the 3-by-3 array on page 2 of A becomes:

```
1 0 0
```

```
0 1 0
0 0 1
```

`A(1,:,:) = ones(1, 3, 2)` becomes

```
A(1,colon(),colon()) = ones(1, 3, 2);
```

As a result of this operation row 1 on both pages of `A` becomes all ones.

Page 1:

```
1 1 1
2 5 8
3 6 9
```

Page 2:

```
1   1  1
11 14 17
12 15 18
```

Finally, `A(2,2,2) = 42` becomes:

```
A(2,2,2) = 42;
```

As a result of this operation the element at `(2,2,2)` changes to the number `42`.

Page 2:

```
10 13 16
11 42 17
12 15 18
```

If the array `A` had more than three dimensions, the subscript would have more than three indices. All of the other types of indexing discussed in this chapter (assigning to entire rows and columns, etc.) work equally well on N-dimensional arrays.

Deleting Elements from an Array

You can use indexing expressions to delete elements from an array. Deletion is a special case of using indexing expressions in assignment statements. Instead of assigning a new value to an element in an array, you assign the null array to a position in the array. The MATLAB C++ Math Library interprets that assignment as a deletion of the element and shrinks the array.

For example, to delete an element from example matrix A, you assign the null array to that element. You create a null array with the `empty()` function.

When you delete a single element from a matrix, the matrix is converted into a row vector that contains one fewer element than the original matrix. For example, when element (8) is deleted from matrix A

```
A(8) = empty();
```

matrix A becomes this row vector with element 8 missing:

```
1 2 3 4 5 6 7 9
```

You can also delete more than one element from a matrix, shrinking the matrix by that number of elements. To retain the rectangularity of the matrix, however, you must delete one or more entire rows or columns. For example,

```
A(2,colon()) = empty();
```

produces this rectangular result:

```
1 4 7
3 6 9
```

Note that the right side of an assignment statement that expresses a deletion is always a call to `empty()`. The left side of the assignment statement must be a valid indexing expression. The null array is applied to each element selected by the subscript.

Note An N-dimensional subscript on the left side of the assignment statement can contain only one scalar, vector, or matrix index. The other indices used in deletion operations must be colon indices. For example, if an array is three-dimensional and you delete row 2, you must delete row 2 from all pages.

Similar to reference and assignment, two-dimensional deletion extends to N dimensions. If A has more than two dimensions, simply specify more than two dimensions as indices in the subscript.

Indexing into Cell Arrays

This section describes how to:

- Reference a cell in a cell array
- Reference a subset of a cell array
- Reference the contents of a cell
- Reference a subset of the contents of a cell
- Index nested cell arrays
- Assign values to a cell array
- Delete elements from a cell array

The examples all use the cell array `N`. `N` contains four cells: a 2-by-2 double array, a string array, an array that contains a complex number, and a scalar array.

cell 1,1
`1 2`
`4 5`
	cell 1,2
	`'Eric'`

cell 2,1 `2-4i`	**cell 2,2** `7`

This MATLAB code creates the array:

```
N{1,1} = [1 2; 4 5];
N{1,2} = 'Eric';
N{2,1} = 2-4i;
N{2,2} = 7;
```

This MATLAB C++ Math Library code creates the array:

```
N.cell(1,1) = vertcat(horzcat(1, 2), horzcat(4, 5));
N.cell(1,2) = "Eric";
N.cell(2,1) = complex(2,-4);
```

```
N.cell(2,2) = 7;
```

Overview

A cell array is a regularly shaped N-dimensional array of *cells*. Each cell is capable of containing any type of MATLAB data, including another cell array. When using cell arrays, you must be careful to distinguish between the data values stored in the cells and the cells themselves, which are data values in their own right.

MATLAB supports two types of indexing on cell arrays. The first, standard indexing, uses parentheses `()` and allows you to manipulate the cells in a cell array. The second, cell array indexing, uses braces `{}` to manipulate the data values stored in the cells.

The MATLAB C++ Math Library supports the same two types of indexing on cell arrays. Standard indexing uses parentheses `()`. Cell array indexing uses `mwArray::cell()` to manipulate the data values stored in the cells. You pass index values to `cell()`.

For example, given the cell array `N`, above, `N{2,2}` in MATLAB and `N.cell(2,2)` in the MATLAB C++ Math Library is the scalar `7`, but `N(2,2)` is a 1-by-1 *cell* array (a single cell) containing the scalar `7`.

Tips for Working with Cell Arrays

- Cell arrays must be regularly shaped. All rows must have the same number of columns, and all columns the same number of rows. This requirement extends into dimensions higher than two, as well. For example, all pages must be the same size in a three-dimensional cell array.
- You can't do arithmetic on a cell. You cannot, for example, write `N(2,2)+1`, which attempts to add one to a cell. However, `N.cell(2,2)+1` works perfectly well, because the cell array indexing returns the contents of cell `(2,2)` rather than the cell itself.
- Cell array indexing follows the same rules as standard indexing. You can use the `colon()` index to refer to multiple rows or columns; you can use vector and matrix indices to extract sub-cell arrays from a cell array.

For simplicity, this section focuses on two-dimensional cell arrays. If `N` were a cell array of higher dimension, the examples would still work on `N`, if you added the appropriate number of dimensions to the indexing expressions.

Referencing a Cell in a Cell Array

To obtain a cell from a cell array, use standard array notation (parentheses) on the right-hand side of the assignment to indicate that you are referencing the *cell itself*, not its contents.

```
c = N(1,2);
```

`c` is a 1-by-1 cell array containing the string array `'Eric'`.

`c = N(1,2)` performs the same operation in MATLAB.

Referencing a Subset of a Cell Array

To obtain a subset of the cells in a cell array, use the `colon()` index or a vector or matrix index to access a group of cells. For example, to extract the second row of the cell array `N`, write this code:

```
B = N(2,colon());
```

The result, `B`, is a 1-by-2 cell array containing the complex number `2-4i` and the integer `7`.

`B = N(2,:)` performs the same operation in MATLAB.

Cell arrays support vector-based (one-dimensional) indexing as well. To extract the first and last elements of `N`, first make a vector `v` that contains the integers `1` and `4`. Use `horzcat()` to construct `v`.

```
B = N(horzcat(1, 4));
```

The result, `B`, is a 1-by-2 cell array that contains a 2-by-2 matrix (element `(1,1)` of `N`) and the scalar `7` (element `(2,2)` of `N`).

`B = N([1 4])` performs the same operation in MATLAB.

Referencing the Contents of a Cell

To obtain the contents of a single cell, use the `mwArray` `cell()` member function to reference the *cell contents* instead of the cell itself. Pass the indices to `cell()`.

```
c = N.cell(1,2);
```

`c` is the string array `'Eric'`.

c = N{1,2} performs the same operation in MATLAB.

Referencing a Subset of the Contents of a Cell

To obtain a subset of a cell's contents, concatenate indexing expressions. For example, to obtain element (2,2) from the array in cell N{1,1}, use an indexing expression that concatenates an index that references the entire *contents* of a cell (using the mwArray cell() member function) with an index that references a portion of those contents (using standard indexing).

```
d = N.cell(1,1)(2,2);
```

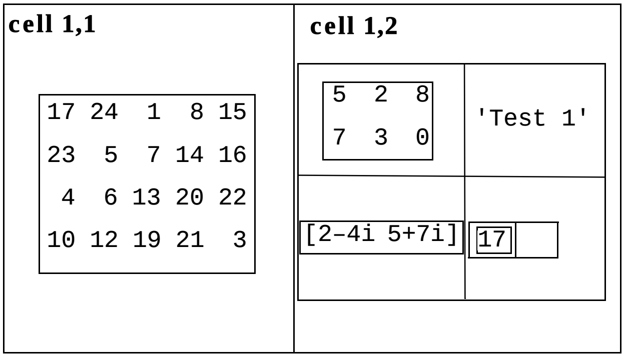
d = N{1,1}(2,2) performs the same operation in MATLAB.

Note that the result d is a scalar array, not a cell array, and equal to 5.

Indexing Nested Cell Arrays

To index nested cells, concatenate subscripts. The first set of subscripts accesses the top layer of cells, and subsequent sets of braces access successively deeper layers of cells.

For example, array A represented in this diagram has three levels of cell nesting: the 1-by-2 cell array itself, the 2-by-2 cell array nested in cell (1,2), and the 1-by-2 cell array nested in cell (2,2).



Indexing into the First Level

To access the 2-by-2 cell array in cell (1,2):

```
A.cell(1,2)
```

In MATLAB `A{1,2}` performs the same operation.

Indexing into the Second Level

To access the 1-by-2 array in position `(2,2)` of cell `(1,2)`:

```
A.cell(1,2).cell(2,2)
```

`A{1,2}{2,2}` in MATLAB performs the same operation.

Indexing into the Third Level

To access the empty cell in position `(2,2)` of cell `(1,2)`:

```
A.cell(1,2).cell(2,2).cell(1,2)
```

`A{1,2}{2,2}{1,2}` in MATLAB performs the same operation.

Assigning Values to a Cell Array

You put a value into a cell array in much the same way that you read a value out of a cell array. In MATLAB, the only difference between the two operations is the position of the cell array relative to the assignment operator: left of the equal sign (=) means assignment, right of the operator means reference. No matter if you're reading or writing values, the indexing operations you use to specify which values to access remain the same. This is true in the MATLAB C++ Math Library as well.

- Use parentheses in indexing expressions for standard array assignments.
- Use `mwArray::cell()` in indexing expressions for cell array assignments.

For example, to assign a vector `[1 2 5 7 11]` to the contents of the cell `(1,2)` of `N`, you write `N{1,2} = [1 2 5 7 11]` in MATLAB and

```
N.cell(1,2) = horzcat(1, 2, 5, 7, 11);
```

in C++ with the MATLAB C++ Math Library.

You could have written the previous assignment in MATLAB as `N(1,2) = { [1 2 5 7 11] }`. The corresponding MATLAB C++ Math Library code is:

```
N(1,2) = cellhcat(horzcat(1, 2, 5, 7, 11));
```

Because this assignment uses parentheses instead of braces, it is an assignment between cells, which means the source array (on the right-hand side of the assignment operator) must be a cell array as well.

Deleting Elements from a Cell Array

Cell arrays follow the same rules as numeric arrays and structure arrays for element deletion. You can delete a single element from a cell array, or an entire dimension element, for example, a row or column of a two-dimensional cell array or a row, column, or page of a three-dimensional cell array. In MATLAB, you delete elements by assigning `[]` to them. In the MATLAB C++ Math Library, you assign the null array.

Deleting a Single Element

In order to delete a single element from an array of any type, you must use one-dimensional indexing. Deleting a single element from a two-dimensional cell array collapses it into a vector cell array. For example, using one-dimensional indexing, `N(2)` refers to element `(2,1)` of `N`. Deleting the `(2,1)` element of `N` (the complex number `2-4i`) produces a three-element cell array. In MATLAB you write `N(2) = []`. See the graphical representation of `N` on page 4-31.

You remove element `(2,1)` from `N` like this:

```
N(2) = empty();
```

Deleting an Entire Dimension

You can delete an entire dimension by using vector subscripting to delete a row or column of cells. Use parentheses within the indexing string to indicate that you are deleting the *cells themselves*.

```
N(2,colon()) = empty();
```

`N(2,:) = []` performs the same operation in MATLAB.

Note `N.cell(2,colon()) = empty();` is an error, because the number of items on the right- and left-hand sides of the assignment operator is not the same. The MATLAB C++ Math Library does not do scalar expansion on cell arrays. If you want to set both cells in the second row of `N` to `[]`, write `N(2,colon()) = cellhcat(empty(), empty())`, thereby assigning a 1-by-2 cell array to another 1-by-2 cell array.

Indexing into MATLAB Structure Arrays

This section describes how:

- To access a field in a structure array
- To access elements within a field of a structure
- To assign a value to a field in a structure array
- To assign a value to an element of a field
- Cell arrays and structure arrays interact
- To delete a field from a structure

The MATLAB C++ Math Library supports two types of indexing on structures. The first, standard indexing, uses parentheses () and allows you to manipulate the structures in a structure array. The second, structure indexing, uses `mwArray::field()` to access the fields in the structure You pass the name of the field to values to `mwArray::field()`.

Overview

A MATLAB structure is very much like a structure in C; it is a variable that contains other variables. Each of the contained variables is called a *field* of the structure, and each field has a unique name.

For example, imagine you were building a database of images. You might want to create a structure with three fields: the image data, a description of the image, and the date the image was created. The following MATLAB code creates this structure:

```
images.image = image1;
images.description = 'Trees at Sunset';
images.date.year = 1998;
images.date.month = 12;
images.date.day = 17;
```

The structure `images` contains three fields: `image`, `description` and `date`. The `date` field is itself a structure, and contains three additional fields: `year`, `month` and `day`. Notice that structures can contain different types of data. `images` contains a matrix (the image), a string (the description), and another structure (the date).

Like standard arrays, structures are inherently array oriented. A single structure is a 1-by-1 structure array, just as the value 5 is a 1-by-1 numeric array. You can build structure arrays with any valid size or shape, including multidimensional structure arrays.

For example, assume you'd like to arrange the images from your database of images in a series of "pages," where each page is three images wide (three columns) and four images tall (four rows). The images might be arranged this way in a photo album or for publication in a journal. The following code demonstrates how you use standard MATLAB indexing to create and access the elements of a 3-by-4-by-n structure array:

```
images(3,4,2).image = image24;
images(3,4,2).description = 'Greater Bird of Paradise';
images(3,4,2).date.year = 1993;
images(3,4,2).date.month = 7;
images(3,4,2).day = 15;
```

For simplicity, the examples in the book focus on two-dimensional structure arrays, but they'd work just as well with structure arrays of any dimension.

Tips for Working with Structure Arrays

- All the structures in a structure array have the same form: every structure has the same fields.
- Adding a field to one structure in a structure array adds it to all the structures in the structure array. Similarly, deleting a field from one structure in the array deletes it from all the structures in the array.
- You can access and modify data stored in the fields of a structure just as you would data stored in an ordinary variable.
- Structure fields are analogous to cell array indices, only they are names rather than numbers.
- Each field in a structure array is an array itself. For example, in the 3-by-4-by-2 example above, the array contains 24 structures. There are 24 images, 24 descriptions, etc., and you can treat each field of the structure as an array of 24 elements. If you typed `images.description`, for example, you'd get a 24-by-1 array of strings containing all the image descriptions in the structure array.

Accessing a Field

The simplest operation on a structure is retrieving data from one of the structure fields. To extract the image field from the second structure in a structure array:

```
image = images(2).field("image");
```

`image = images(2).image` performs the same operation in MATLAB.

Accessing the Contents of a Structure Field

A structure field may contain another array. By performing additional indexing operations, you can access the data stored in that array. You must specify the field name as an argument to `mwArray::field()` and then apply the appropriate type of indexing to the data in that field:

- Use array subscripting if the field contains an array.
- Use cell array subscripting if the field contains a cell array.

For example, this code retrieves the first row of the image in the third structure:

```
n =x(3).field("image")(1,colon());
```

`n = x(3).image(1,:)` performs the same operation in MATLAB.

Assigning Values to a Structure Field

To assign an initial value to a field (creating the field if it doesn't exist) or to modify the value of an existing field, use structure array indexing on the left-hand side of the assignment operator. For example, to change the description field of the seventeenth image, you'd write this code:

```
images(17).field("description") = "Ferris Wheel";
```

`images(17).description = 'Ferris Wheel'` performs the same operation in MATLAB.

Assigning Values to Elements in a Field

You can also modify array data contained in a structure field. You must pass the field name to `mwArray::field()` and the type of indexing to perform on the contained array. For example, this code replaces a 3-by-3 subarray of the image data of the ninth image, with the data in the 3-by-3 array x. You might do this as part of some image processing operation.

```
images(9).field("image")(colon(1,3),colon(2,4)) = x;
```

`images(9).image(1:3,2:4) = x` performs the same operation in MATLAB.

Referencing a Single Structure in a Structure Array

To access a single structure within the structure array, use standard array notation. For example, to reference the forty-second image structure in a structure array, use this code:

```
B = images(42);
```

`B = images(42)` performs the same operation in MATLAB.

Referencing into Nested Structures

Structures can contain other structures. For example, the image structure used in these examples contains a `date` structure. To retrieve data from nested structures, nest calls to `mwArray::field()`.

```
y = images(2).field("date").field("year");
```

`y = images(2).date.year` performs the same operation in MATLAB.

Note You can only reference or assign to single instances of nested structures. Though you might expect this MATLAB C++ Math Library code `y = images.field("date").field("year")` to set `y` to the array of years in the `date` field of the `images` structure array, this code generates an error because the result of `images.field("date")` is a structure array rather than a single structure.

Accessing the Contents of Structures Within Cells

Cell arrays can contain structure arrays and vice-versa. Accessing a structure stored in a cell array is very similar to accessing a structure stored in a regular variable; you just need to extract it from the cell array first. You use `mwArray::cell()` to extract the cell array.

Assume the cell array c contains a three-element structure array of images.

You can also combine cell array and standard indexing to access a single field of a single structure:

```
second_date = c.cell(1)(2).field("date");
```

`second_date = c{1}(2).date` performs the same operation in MATLAB. In this case, the result is a single date structure.

Deleting Elements from a Structure Array

There are three kinds of deletion operations you can perform on a structure array.

You can delete:

- An entire structure from the array
- A field from all the structures in the array
- Elements from an array contained in a field

Deleting a Structure from the Array

To delete an entire structure from a structure array, assign the null array to that structure. For example, if you have a three-element array of image structures, you can delete the second image structure like this:

```
images(2) = empty();
```

`images(2) = []` performs the same operation in MATLAB. The result is a two-element array of image structures.

Deleting a Field from All the Structures in an Array

To delete a field from all the structures in the array, use `rmfield()`. For example, you can remove the description field from an array of image structures, with this code:

```
images = rmfield(images, "description");
```

`images = rmfield(images, ’description’)` performs the same operation in MATLAB.

Note `rmfield()` does *not* allow you to remove a field of a nested structure from a structure array. For example, you cannot remove the `day` field of the nested `date` structure with `rmfield(images.field("date"), "day")`. This is an *error* in the MATLAB C++ Math Library and in MATLAB.

Deleting an Element from an Array Contained by a Field

To delete an element from an array contained by a field, assign the null array to the indexing expression. For example, to remove the fifth column of the image in the third image structure:

```
images(3).field("image")(colon(),5) = empty();
```

`images(3).image(:,5) = []` performs the same operation in MATLAB.

Indexing Techniques

The following sections describe some common indexing task and how to accomplish them.

Duplicating a Row or Column

You can make duplicate copies of an array row or column in two different ways: an intuitive way and a short way.

Assume that you want to make a matrix that consists of four copies of the first row of a 5-by-5 matrix, for example, the matrix returned by `magic(5)`:

```
17    24     1     8    15
23     5     7    14    16
 4     6    13    20    22
10    12    19    21     3
11    18    25     2     9
```

The Intuitive Solution

For the straightforward approach, you use the `vertcat()` or `horzcat()` functions in the MATLAB C++ Math Library. (In MATLAB you would use the concatenation operator `[]`.) This approach requires two lines of code (one assignment and one concatenation) or one long line:

```
mwArray A = magic(5);
mwArray B = A(1,colon());
mwArray C = vertcat(B, B, B, B);
```

The code makes `C` into a 4-by-5 matrix, using two lines of code. (Don't count the line that declares `A`.) First, the first row of `A` is assigned to `B`. Then `vertcat()` concatenates B four times into `C`, producing this result:

```
17    24    1     8    15
17    24    1     8    15
17    24    1     8    15
17    24    1     8    15
```

The Shortcut

You can accomplish the same task with a single short line.

```
mwArray A = magic(5);
```

```
mwArray C = A(ones(1,4), colon());
```

This code produces the same matrix as the previous code fragment, but does not require the declaration of the intermediate matrix B. The ones() function creates a vector of four 1's, which as a subscript, selects the first row in matrix A four times.

You can use this trick to duplicate columns instead of rows by switching the positions of the calls to ones() and colon():

```
mwArray C = A(colon(), ones(1,4));
```

This creates a 5-by-4 matrix containing duplicates of the first column of A:

```
17    17    17    17
23    23    23    23
 4     4     4     4
10    10    10    10
11    11    11    11
```

Concatenating Subscripts

In MATLAB, you apply an index operation to a variable. You cannot apply an index to the result of a function call or to the result of an arithmetic operation, without first assigning the result to an array variable.

In C++, however, you *can* apply an index to any object of type mwArray or of a type that can be automatically converted into an mwArray, including mwArray results from function calls, arithmetic operations and indexing operations. Being able to perform an indexing operation without having to declare a temporary variable first is very convenient.

This is a notational convenience only; your code does not run faster.

Applying a Subscript to the Result of a Function Call

You can easily compose function calls using this technique. Applying a subscript to the result of a function call lets you extract a subarray from the result and pass that result directly to a second function, without having to assign the result to a variable first.

For example, this code extracts a 3-by-3 array from a 10-by-10 magic square, and passes the 3-by-3 array to sqrt().

```
mwIndex i = ramp(4,6);
```

```
mwArray A = sqrt(magic(10)(i,i));
```

Applying a Subscript to the Result of an Arithmetic Operation

You can apply a subscript to the result of an arithmetic operation. For example, this code multiplies two random 4-by-4 arrays, `A` and `B`, and extracts the `(2,2)` element of the result into a double precision floating-point scalar, `x`.

```
mwArray A = rand(4), B = rand(4);
double x = (A * B)(2,2);
```

By moving the calls to `rand()` to the second line, you can rewrite this example in one line:

```
double x = (rand(4) * rand(4))(2,2);
```

This technique works with logical operations as well.

C++ and MATLAB Indexing Syntax

The table below summarizes differences between C++ and MATLAB standard array indexing syntax. Although the MATLAB C++ Math Library provides the same functionality as the MATLAB interpreter, the syntax of some operations is slightly different. In particular, you must use the `colon()` function rather than the colon operator.

Though not listed here, you must use `mwArray.cell()` rather than `{}` and `mwArray::field()` rather than the period (`.`) that references a structure field.

Note For the examples in the table, matrix `X` is set to the 2-by-2 matrix `[ 4 5 ; 6 7 ]`, a different value from the 3-by-3 matrix `A` in the previous sections.

Example Matrix X

```
4 5
6 7
```

Table 4-1: MATLAB/C++ Indexing Expression Equivalence

Description	MATLAB Expression	C++ Expression	Result
Extract `1,1` element	`X(1,1)`	`X(1,1)`	4
Extract first element	`X(1)`	`X(1)`	4
Extract third element	`X(3)`	`X(3)`	5
Extract all elements into a column vector	`X(:)`	`X(colon())`	4 6 5 7
Extract first row	`X(1,:)`	`X(1,colon())`	4 5
Extract second row	`X(2,:)`	`X(2,colon())`	6 7

Table 4-1: MATLAB/C++ Indexing Expression Equivalence (Continued)

Description	MATLAB Expression	C++ Expression	Result
Extract first column	`X(:,1)`	`X(colon(), 1)`	4 6
Extract second column	`X(:,2)`	`X(colon(), 2)`	5 7
Replace first element with 9	`X(1) = 9`	`X(1) = 9`	9 5 6 7
Replace first row with `[ 11 12 ]`	`X(1,:) = [ 11 12 ]`	`X(1,colon())=` `horzcat(11,12);`	11 12 6 7
Replace element `2,1` with `9`	`X(2,1) = 9`	`X(2,1) = 9;`	4 5 9 7
Replace elements `1` and `4` with `8` (one-dimensional indexing)	`X([1 4]) = [ 8 8 ]`	`X(horzcat(1,4))=` `horzcat(8, 8);`	8 5 6 8

The mwIndex Class

mwArray overloads the () operator for MATLAB-like indexing. The mwArray::operator() functions can accept mwIndex objects. An mwIndex can represent a single integer, a sequence of integers specified by a tuple (start, step, stop), or an arbitrary vector of integers. Array subscripts are always integers; the MATLAB C++ Math Library truncates floating-point numbers to integers in indexing expressions.

You can create mwIndex objects in several ways. The way most familiar to MATLAB users is the colon() function. An mwIndex constructor exists for each form of the colon() function, as this table demonstrates.

Note The default mwIndex constructor (entry one in the table below) produces an mwIndex object representing : (colon).

Table 4-2: mwIndex Class and the colon() Function

Description	MATLAB Expression	C++ colon() Equivalent	mwIndex Constructor
All elements of X	X(:)	X(colon())	mwIndex k; X(k)
First 10 elements of row 1	X(1,1:10)	X(1,colon(1,10))	mwIndex k(1,10); X(k)
Elements 2,4,6,8,10 of X	X(2:2:10)	X(colon(2,2,10))	mwIndex k(2,2,10); X(k)

Programming Efficient Indices

If you use the same index repeatedly, store it in an `mwIndex` variable instead of creating it each time. The cost of creating `mwIndex` objects is low, but measurable. If you are bothered by having to type `colon()` too frequently, you can create an `mwIndex` variable with a shorter name, `mwIndex all`, for example, and use it instead of the `colon()` function.

5

Calling Library Functions

Overview

The MATLAB C++ Math Library includes over 400 functions. Every routine in the library works the same way as its corresponding routine in MATLAB. This chapter describes the calling conventions that apply to the library functions, including how the C++ interface to the functions differs from the MATLAB interface. Once you understand the calling conventions, you can translate any call to a MATLAB function into a C++ call.

This chapter includes information about passing a function as an argument to a MATLAB function or a function of your own creation.

Chapter 11 contains a listing of all the routines in the MATLAB C++ Math Library. For complete reference information about the library functions, including the list of arguments and return value for each function, see the *MATLAB C Math Library Reference*. Each function reference page includes a link to the documentation for the MATLAB version of the function. "Accessing Online Reference Documentation" on page 1-7 describes how to use the Help system.

How to Call C++ Library Functions

The following sections use the `cos()`, `tril()`, `find()`, and `svd()` functions to demonstrate how to translate a MATLAB call to a function into a C++ Math Library call. Each of the functions demonstrates a different aspect of the calling conventions, including what data type to use for C++ input and output arguments, how to handle optional arguments, and how to handle MATLAB's multiple output values in C++. Specifically, the topics covered include:

- "One Result and Only Required Input Arguments" on page 5-3
- "Passing Optional Input Arguments" on page 5-3
- "Passing Optional Output Arguments" on page 5-4
- "Passing Optional Input and Output Arguments" on page 5-5
- "Passing Any Number of Inputs" on page 5-6
- "Passing Any Number of Outputs" on page 5-8
- "Summary of Library Calling Conventions" on page 5-10
- "Example Program: Calling Library Functions (ex2.cpp)" on page 5-12

One Result and Only Required Input Arguments

For many functions in the MATLAB C++ Math Library, the translation from interpreted MATLAB to C++ is simple. For example, in interpreted MATLAB, you invoke the cosine function, `cos()`, like this

```
Y = cos(X)
```

where both X and Y are arrays.

Using the MATLAB C++ Math Library, you invoke cosine in exactly the same way

```
Y = cos(X);
```

where both `X` and `Y` are `mwArray` objects.

Passing Optional Input Arguments

Some MATLAB functions take optional input and output arguments. `tril()`, for example, which returns the lower triangular part of a matrix, takes either one or two input arguments. If present, the second input argument, `k`, indicates which diagonal to use as the upper bound; `k=0` indicates the main diagonal and is the default if no `k` is specified. In interpreted MATLAB you invoke `tril()` either as

```
L = tril(X)
```

or

```
L = tril(X, k)
```

where `L`, `X`, and `k` are matrices. `k` is a 1-by-1 array.

The MATLAB C++ Math Library contains two versions of the `tril()` function. The first version takes one argument; the second takes two arguments. The two ways to call the MATLAB C++ Math Library versions of `tril()` are exactly the same as the two ways you can call `tril()` in interpreted MATLAB

```
L = tril(X);
```

and

```
L = tril(X, k);
```

where `L`, `X` and `k` are `mwArray` objects.

Passing Optional Output Arguments

MATLAB functions may also have optional or multiple output arguments. For example, you invoke the `find()` function, which locates nonzero entries in matrices, with one, two, or three output arguments:

```
k = find(X);
[i, j] = find(X);
[i, j, v] = find(X);
```

In interpreted MATLAB, `find()` returns one, two or three values. In C++, no function can return more than one value. Therefore, the additional output arrays are passed to `find()` in the argument list. Output arguments are always pointers to `mwArray` objects, (`mwArray*` variables), and they always appear before input arguments in the parameter list.

To accommodate all the combinations of output arguments, there are three overloaded versions of `find()` in the MATLAB C++ Math Library. Using the MATLAB C++ Math Library, you call `find()` like this:

```
k = find(X);
i = find(&j, X);
i = find(&j, &v, X);
```

k, i, j, v, and X are mwArray objects. You do not need to preallocate k, i, j, or v; when you declare them as mwArray objects, they are appropriately initialized.

Note how easy it is to distinguish input variables from output variables; an ampersand (&) always precedes each output variable. In C++, the & operator, when placed in front of an array, computes the address of, or pointer to, that array. All of the arguments with & placed in front of them are output arguments, corresponding to the variables on the left-hand side of the MATLAB expression.

The general rule for multiple output arguments: use the function return value, an mwArray, as the first output argument; pass all additional output arguments into the function as mwArray* parameters. By convention, output arguments always come first, followed by input arguments. Putting the output arguments first may surprise some C++ programmers because it prevents the use of default values for optional arguments. However, this ordering is more natural for MATLAB programmers, since it keeps the output arguments, which in MATLAB would be on the left-hand side of the assignment operator, as close to the left-hand side as possible.

Passing Optional Input and Output Arguments

Finally, a MATLAB function may have both optional input and optional output arguments. The MATLAB C++ Math Library provides multiple overloaded functions to implement the various calls. The svd() function, for example, has three forms. The first takes one input and returns one output. The second takes one input and returns three outputs. The third takes two inputs and returns three outputs. Note that the return value counts as one output.

```
S = svd(X);
U = svd(&S, &V, X);
U = svd(&S, &V, X, Zero);
```

U, S, V, X, and Zero are all mwArray objects.

Passing Any Number of Inputs

Some MATLAB functions accept any number of input arguments. In MATLAB these functions are called `varargin` functions. When the variable `varargin` appears as the last input argument in the definition of a MATLAB function, you can pass any number of input arguments to the function, starting at that position in the argument list.

MATLAB takes the arguments you pass and stores them in a cell array, which can hold any size or kind of data. The `varargin` function then treats the elements of that cell array exactly as if they were arguments passed to the function.

Whenever you see an ellipsis (...) at the end of the input argument list in a MATLAB syntax description, the function is a `varargin` function. For example, the syntax for the MATLAB function `cat` includes the following specification in the online *MATLAB Function Reference*.

```
B = cat(dim,A1,A2,A3,A4,...)
```

`cat` accepts any number of arguments. The `dim` and `A1` arguments to `cat` are required. You then concatenate any number of additional arrays along dimension `dim`. For example, this call concatenates six arrays along the second dimension

```
B = cat(2,A1,A2,A3,A4,A5,A6)
```

Because the C++ language does not support functions that accept variable-length argument lists, the MATLAB C++ Math Library supports MATLAB `varargin` functions through overloading and the `mwVarargin` class.

In the MATLAB C++ Math Library, you invoke the `cat` function like this if you are passing 32 or fewer array arguments. The call looks just like the MATLAB call

```
B = cat(2,A1,A2,A3,A4,A5,A6);
```

where `B` and the six `A` matrices are `mwArray` objects.

However, if you need to pass more than 32 arguments to a `varargin` function in the MATLAB C++ Math Library, you must construct an `mwVarargin` object that you pass as the first argument following any required or optional input arguments. The `mwVarargin` object stores up to 32 input arguments, the first of which can be another `mwVarargin` object, allowing you to create any length input argument list.

Constructing an mwVarargin Object

MATLAB C++ Math Library functions that take a variable number of input arguments have one mwVarargin argument followed by 31 additional mwArray arguments:

- If you pass 32 or fewer arguments, you can ignore the mwVarargin parameter and simply pass a series of mwArrays as with any other function.
- If you need to pass more than 32 inputs, you must construct an mwVarargin object and pass it as the mwVarargin parameter.

The mwVarargin constructor has the standard varargin parameter list: one mwVarargin argument followed by 31 additional mwArray arguments. The mwVarargin constructors can be nested enabling you to pass an unlimited number of inputs.

The inputs used to construct the mwVarargin argument appear first on the argument list for the function, followed by the remaining 31 inputs. It is not necessary to fill out the mwVarargin constructor parameter list. The arguments can be distributed between the mwVarargin constructor and the remaining 31 arguments.

For example, the library function horzcat() is a varargin function that demonstrates the standard varargin parameter list. Its function prototype is

```
mwArray horzcat(const mwVarargin &in1=mwArray::DIN,
                 const mwArray &in2=mwArray::DIN,
                 .
                 .
                 .
                 const mwArray &in32=mwArray::DIN);
```

To pass 90 inputs to the horzcat function, make this call:

```
horzcat(mwVarargin(mwVarargin(p1,p2,...,p32), p33, ..., p63),
        p64, ..., p90);
```

The first 32 arguments are passed to an mwVarargin constructor that is nested as the first argument to another mwVarargin constructor. The next 31 arguments (p33 through p63) are passed as mwArray arguments to the mwVarargin object that is the first argument to horzcat(). The remaining arguments (p64 through p90) are passed as additional mwArray arguments to the function.

Note that the ... represent omitted arguments in the series and are not part of the actual function prototype or function call.

Note If a function takes any required output arguments, an `mwVarargout` argument, or any required or optional input arguments, these arguments precede the first `mwVarargin` argument in the list of arguments.

Passing Any Number of Outputs

Some MATLAB functions return any number of outputs. In MATLAB these functions are called `varargout` functions. When the variable `varargout` appears as the last output argument in the definition of a MATLAB function, that function can return any number of outputs, starting at that position in the argument list.

When you call a `varargout` function in the interpreted MATLAB environment, MATLAB takes the arguments you pass and stores them in the cell array called `varargout`. A cell array can hold any size or kind of data. Because the arguments are output arguments, they don't need to exist yet. The MATLAB function accesses the varying number of arguments passed to it through the cell array.

Whenever you see an ellipsis (`...`) within the output argument list of a MATLAB syntax description, the function is a `varargout` function. For example, this syntax in the online *MATLAB Function Reference* specifies a version of the MATLAB function `size` that returns a variable number of outputs depending on the number of dimensions in the array passed to it.

```
[M1,M2,M3,...,MN] = size(X)
```

If the dimensionality of the input argument `X` is 2, `size` returns the length of the first dimension in the first output value and the length of the second dimension in a second output value; if the dimensionality is 4, it returns four lengths.

For example, if the input array, `X`, has four dimensions, this code retrieves the length of each dimension:

```
[d1,d2,d3,d4] = size(X)
```

In the MATLAB C++ Math Library you invoke the same call by constructing an mwVarargout object that contains the arguments you are passing. The first output argument is always the return value

```
size(mwVarargout(d1, d2, d3, d4), X);
```

where X and d1, d2, d3, and d4 are mwArray objects. Note that you do not pass the address of the mwArray objects to the mwVarargout constructor even though they are output arguments.

Constructing an mwVarargout Object

MATLAB C++ Math Library functions that produce a variable number of outputs have an mwVarargout parameter as their last output argument.

In order to retrieve the varargout outputs from the function, you need to construct an mwVarargout object. You pass the variables to which the outputs will be assigned to the mwVarargout constructor and then pass the mwVarargout object as the last output argument to the function.

The arguments to the mwVarargout constructor differ from normal output arguments in two ways. When constructing an mwVarargout object:

- You pass the array itself, not a pointer to the array, to the constructor.
- You can pass indexed expressions as inputs. Anything that can appear on the left-hand side of an assignment can appear as an argument to the mwVarargout constructor.

For example, this code demonstrates a call to the M-function size, which takes a variable number of output arguments and a single input argument. The prototype for size() in C++ specifies an mwVarargout object, as its first parameter, and one or two input arguments. The call to size() in C++ corresponds to the call in M.

M code:

```
[x, y(2,3), z{:}] = size(m)
```

C++ prototype:

```
mwArray size(mwVarargout varargout,
             const mwArray &in1,
             const mwArray &in2=mwArray::DIN);
```

C++ call:

```
size(mwVarargout(x, y(2,3), z.cell(colon())), m);
```

Note that the function `size()` takes no other required output arguments besides a `varargout` argument. It is called a "pure" `varargout` function. In pure `varargout` functions, the return value of the function is the same as the value assigned to the first element of the `mwVarargout` object, in this case the variable `x`. When calling a pure `varargout` function, you do not need to assign the output of the function to the first output argument explicitly; simply pass it to the `mwVarargout` constructor. For all functions in the MATLAB C++ Math Library, if the first argument is `mwVarargout`, the function is pure `varargout`.

If other output arguments precede the `mwVarargout` parameter, then the return value is not part of the `mwVarargout` object and must be explicitly assigned to a return value.

Summary of Library Calling Conventions

Several rules express the formal mapping between the MATLAB and C++ calling conventions:

1. If there is only one output argument, the syntax of the MATLAB C++ Math Library is identical to the interpreted MATLAB syntax.

   For example, the MATLAB statement `A = eig(C);` translates to the identical C++ statement `A = eig(C);`.

2. If there is more than one output argument, the first output argument becomes the function return value. The others are passed as output arguments and are each prefixed with an `&`. They precede the input arguments in the argument list.

   For example, the MATLAB function call `[ U, S, V ] = svd(X)` has three output arguments, `U`, `S`, and `V`, and one input argument `X`. The corresponding call in C++ is `U = svd(&S, &V, X)`. Note that the two output arguments in the argument list *must* be prefixed with an `&`, as the C++ library requires output arguments to be passed by reference.

Tip You can also slide the left-hand MATLAB variables to the right side (prefixing them with &) until only one variable remains on the left-hand side.

---

3 If there is a variable-length output argument list, you must construct an `mwVarargout` object that can represent any number of arguments. The constructor takes 32 arguments, the first of which can be another `mwVarargout` object, allowing you to create any length output argument list.

4 If there are more than 32 input arguments, you must construct an `mwVarargin` object that can itself represent 32 arguments, the first of which can be another `mwVarargin` object, allowing you to create any length argument list.

The following table summarizes the mapping between interpreted MATLAB functions and the same functions in the MATLAB C++ Math Library.

Table 5-1: MATLAB and C++ Function Calling Conventions

MATLAB Calling Sequence	C++ Calling Sequence	Input/Output Count
`A = eig(C);`	`A = eig(C);`	one input one output
`L = tril(X, k);`	`L = tril(X, k);`	two inputs one output
`[ A, B ] = eig(C);`	`A = eig(&B, C);`	one input two outputs
`[ U, S, V ] = svd(X);`	`U = svd(&S, &V, X);`	one input three outputs
`[ U, S, V ] = svd(X, 0);`	`U = svd(&S, &V, X, 0);`	two inputs three outputs

Table 5-1: MATLAB and C++ Function Calling Conventions (Continued)

MATLAB Calling Sequence	C++ Calling Sequence	Input/Output Count
B = cat(2,A1,A2,A3,A4,A5,A6);	B = cat(2,A1,A2,A3,A4,A5,A6);	seven inputs one output
[d1,d2,d3,d4] = size(X);	size(mwVarargout(d1, d2, d3, d4), X);	one input four outputs

Exceptions to the Calling Conventions

The load() and save() functions do not follow the standard calling conventions for the library. For information about load() and save(), see “Importing and Exporting MAT-File Data” in Chapter 8.

Example Program: Calling Library Functions (ex2.cpp)

This example demonstrates how to call different versions of the same library function and how to pass optional input and output arguments. The example uses the svd() function.

You can find the code for this example in the <matlab>/extern/examples/cppmath directory on UNIX systems and in the <matlab>\extern\examples\cppmath directory on PCs, where <matlab> represents the top-level directory of your installation. See “Building C++ Applications” on page 1-13for information about building and running the example program.

In MATLAB, there is one svd() function that you can call with varying numbers of input and output arguments. The MATLAB version of svd() counts the number of arguments passed to it and performs a different calculation for each valid combination of input and output arguments.

An ordinary C++ function cannot count its arguments. It always requires the same number of inputs and outputs each time you call it. However, you can declare multiple C++ functions with the same name as long as the argument lists for the functions are different. This is called overloading a function. Argument lists differ if they contain different numbers of arguments or if the types of the arguments are different.

There are three ways to call the svd() function in MATLAB. Therefore there are three overloaded svd() functions in C++, each corresponding to one of the ways you can call svd() in MATLAB. This example demonstrates how to call each of the overloaded svd() functions.

Refer to the online *MATLAB C++ Math Library Reference* for an explanation of svd(). "Accessing Online Reference Documentation" on page 1-7 describes how to access the Help Desk.

In this example, note the following:

- Each MATLAB function that can be called with varying numbers of arguments corresponds to a set of overloaded functions in the MATLAB C++ Math Library.
- Place all output arguments before any input arguments in the parameter list.
- The function return value corresponds to the first output argument.
- Always pass an input argument to a function as an mwArray object.
- Always pass an output argument as a pointer to mwArray object.
- You may omit optional arguments from the parameter list. Placeholder or default values are not necessary.
- MATLAB C++ Math Library functions never modify their input arguments and always modify their output arguments.
- Call C++ constructors where possible, for efficiency.

```
   // ex2.cpp

   #include <stdlib.h>
①  #include "matlab.hpp"

②  static double data[] = { 1, 3, 5, 7, 2, 4, 6, 8 };

   int main(void)
   {
       // Create the input matrix.
③      mwArray X(4, 2, data);
       mwArray U, S, V;

       // Compute the singular value decomposition of the matrix.
       cout << "One input, one output: " << endl ;
④      cout << "S = " << svd(X) << endl;

       // Pass in optional output arguments.
⑤      U = svd(&S, &V, X);

       cout << "One input, three outputs: " << endl;
       cout << "U = " << U << "S = " << S << "V = " << V << endl;

       // Pass in optional input argument.
⑥      U = svd(&S, &V, X, 0.0);

       cout << "Two inputs, three outputs: " << endl;
       cout << "U = " << U << "S = " << S << "V = " << V;

       return(EXIT_SUCCESS);
   }
```

The numbered items in the list below correspond to the numbered sections of code example:

1 Include header files. `matlab.hpp` declares the MATLAB C++ Math Library's data types and functions. `matlab.hpp` includes `iostream.h`, which declares the input and output streams `cin` and `cout`. `stdlib.h` contains the definition of `EXIT_SUCCESS`.

2 Declare the static C++ array that initializes the `svd()` input matrix in the `main()` routine. The MATLAB C++ Math Library requires that the numbers in this array be in column-major order. See “Example Program: Creating Arrays and Array I/O (ex1.cpp)” in Chapter 3 for more information.

3 Call a C++ constructor to declare and initialize the matrix `X` to a four-row, two-column matrix. In your code, use C++ constructors whenever possible, as they are more efficient than a declaration followed by an assignment.

4 Call the simplest of the three `svd()` functions. This function takes one input matrix and produces one output matrix. Because C++ allows multiple functions with the same name to co-exist as long as their argument lists are different (this is called overloading a function), calling the simplest form of `svd()` does not require passing in extra NULL arguments as it would in the MATLAB C Math Library.

5 Call the second `svd()` function. This function takes one input and produces three outputs. Because C++, unlike MATLAB, does not allow multiple return values, you pass the extra outputs into the `svd()` function as output arguments. Output arguments are modified by the function to contain the appropriate results.

   Just as input arguments are uniformly passed as `const mwArray` references, output arguments are always passed as pointers to `mwArray` objects. In C++, applying the `&` (address-of) operator to an object produces a pointer to that object.

6 Call the third `svd()` function, which takes two inputs and produces three outputs. The second input is an optional argument. Note again the convenience of C++ function overloading. Without overloading, this optional input argument would have appeared as a `NULL` value in the argument lists of the other calls to `svd()`.

Output

The program produces the following output. See “Using Array Stream I/O” in Chapter 8 for details on the array input and output format.

```
One input, one output:
S =   1.0e+01 *
```

```
  [
    1.42691 ;
    0.06268
  ]

 One input, three outputs:
U =   [
    0.15248   0.82265  -0.39450  -0.37996 ;
    0.34992   0.42138   0.24280   0.80066 ;
    0.54735   0.02010   0.69791  -0.46143 ;
    0.74479  -0.38117  -0.54621   0.04074
  ]

S =   1.0e+01 *

  [
    1.42691   0.00000 ;
    0.00000   0.06268 ;
    0.00000   0.00000 ;
    0.00000   0.00000
  ]

V =   [
    0.64142  -0.76719 ;
    0.76719   0.64142
  ]

 Two inputs, three outputs:
U =   [
    0.15248   0.82265 ;
    0.34992   0.42138 ;
    0.54735   0.02010 ;
    0.74479  -0.38117
  ]

S =   1.0e+01 *

  [
    1.42691   0.00000 ;
    0.00000   0.06268
```

```
  ]

V =   [
    0.64142  -0.76719 ;
    0.76719   0.64142
  ]
```

How to Call Operators

Many of the operators in MATLAB have operator equivalents in C++. The syntax for these C++ operators is identical to that of their MATLAB counterparts, and you call them directly as operators.

In addition, every operator in MATLAB is mapped directly to a function in the MATLAB C++ Math Library. For MATLAB operators that do not have operator equivalents in C++, determine the name of the function that corresponds to the operator and then call the function as explained above.

The section “Operators” in Chapter 11 lists the MATLAB operators and the corresponding MATLAB C++ Math Library functions.

Example – Passing Functions As Arguments (ex3.cpp)

This example covers advanced material. You only need to read this section if you're using a MATLAB C++ Math Library function that requires another function as an argument.

You can find the code for this example in the `<matlab>/extern/examples/cppmath` directory on UNIX systems and in the `<matlab>\extern\examples\cppmath` directory on PCs, where `<matlab>` represents the top-level directory of your installation. See "Building C++ Applications" on page 1-13 for information about building and running the example program.

Certain functions in the MATLAB C++ Math Library, for example, `fmins()` and `fzero()`, require user-supplied functions as arguments. `fmins()` and the functions like it are called "function-functions," because they operate on functions rather than arrays. This example demonstrates how to write a function that a function-function can call.

The library supports two methods of registering your function with the MATLAB C++ Math Library: the first, and easiest, uses the `feval` macros; the second requires that you write a thunk function and define and populate a local table that identifies your function for the library. The macro method performs these tasks for you.

Both methods are presented in this example. The macros support registration of the most common types of functions. You only need to use the manual, nonmacro method in certain special cases (detailed below). Read the step-by-step, nonmacro version if you want to understand in detail how the MATLAB C Math Library function `mlfFeval()` executes the functions passed to it.

The MATLAB C Math Library forms the foundation for the MATLAB C++ Math Library. For the most part, the MATLAB C Math Library provides its services transparently, but there are a few places where its interface is visible.

To execute a function passed to a function-function, the C++ Math Library calls the C Math Library function `mlfFeval()`. `mlfFeval()` calls a thunk function that actually executes the function passed to it. That thunk function must have a C interface along with the table that identifies your function to the library. Refer to the example "Passing Functions as Arguments" in the *MATLAB C*

Math Library User's Guide for more information on how `mlfFeval()` and thunk functions work.

Note You don't need to use the `feval` macros if you want a function-function or `feval()` to execute a MATLAB C++ Math Library function. A thunk function and an entry in the built-in table already exist for the library functions. In addition, if you're using the MATLAB Compiler, it automatically generates all the code you need.

In this example, note the following:

- `fmins()`, `fzero()`, and the other function-functions in the MATLAB C++ Math Library take a function name as an input argument. The function-function executes that function. The function can be one you've written or a library function.
- In C++, a function pointer serves the same purpose as a function name serves in interpreted MATLAB: both enable you to call a function.
- The MATLAB C Math Library forms the foundation for the MATLAB C++ Math Library. Its function `mlfFeval()` executes the functions passed to the C++ Math Library function-functions.
- A thunk function translates the interface required by one function into the interface required by another. In the MATLAB C++ Math Library, it translates the C `mlfFeval()` interface into the C++ interface of a function to be executed.
- The `DECLARE_FEVAL_TABLE` macros provide an easy way to register one of your functions with the function `mlfFeval()`. If you use the macros, you don't have to write a thunk function.
- If you don't use the `feval` macros, you must provide a thunk function that conforms to the MATLAB C Math Library interface rather than the MATLAB C++ Math Library interface. When you write a thunk function, you must follow several guidelines. In particular, you must be careful not to delete or free any of the data passed into the function and to return a newly allocated array from the function.
- In the MATLAB C Math Library interface, an array is a pointer to an `mxArray` structure.

- Passing arrays represented as `mxArray` pointers to functions in the MATLAB C++ Math Library or to a function you've written in C++ requires explicit conversion of the `mxArray` pointers to `mwArray` objects. The default conversion *does the wrong thing*; you *must* explicitly specify that the `mxArray` data not be freed when the `mwArray` object goes out of scope.
- The `mwArray` member function `FreezeData()` modifies the `mwArray` so that it does not free the `mxArray` it contains. However, the function is dangerous and invites memory leaks and memory protection errors. Use it with great care, exactly as outlined in this example.

Using the feval Macros

Use the `feval` macros to register a function that you've written for execution by a function-function or `feval()`. The macros register any function that takes a combination of 0 to 8 input arguments and 1 to 5 output arguments. If you need to register a function that takes more than 8 inputs or more than 5 outputs, you cannot use the `feval` macros; you must write your own thunk function and manually construct an `feval` function table.

Note You cannot register a function that has been overloaded. In addition, the arguments to the function being registered must be of type `mwArray` or `mwArray *`, not type `const mwArray&`.

The functions `func1()` and `main()` are the same in both versions of this example. The `feval` macros replace the thunk function, a `typedef`, a `mlfFuncTabEnt` declaration, and the `feval_init` class that you'll find in the nonmacro version.

```
    // Example 3, macro version

    #include <stdlib.h>
(1) #include "matlab.hpp"

(2) mwArray func1(mwArray x)
    {
      // one argument test function
      return(times(realsqrt(x), reallog(x)));
    }

(3) DECLARE_FEVAL_TABLE
      FEVAL_ENTRY(func1)
    END_FEVAL_TABLE

    int main(void)
    {
        try {
(4)         cout << fmins("func1", 0.25) << endl;
        }
        catch (mwException &ex)
        {
            cout << ex;
        }
      return(EXIT_SUCCESS);
    }
```

The numbered items in the list below correspond to the numbered sections of code example:

1 Include header files. `matlab.hpp` declares the MATLAB C++ Math Library's data types and functions. `matlab.hpp` includes `iostream.h`, which declares the input and output streams `cin` and `cout`. `stdlib.h` contains the definition of `EXIT_SUCCESS`.

2 Declare `func1()`. The name of this function is subsequently passed to `fmins()`. During the execution of `fmins()`, control passes into the MATLAB C Math Library, which calls `func1()`.

`func1()` computes the natural logarithms and the square roots of the elements in the input matrix and multiplies them together. The two functions, `reallog()` and `realsqrt()` guarantee that their outputs are noncomplex matrices, i.e., matrices that have only real (no imaginary component) elements. Note that this computation means nothing mathematically.

3 Use the `feval` macros to register `func1()` as a function that can be executed by a function-function or `feval()`. Begin the table with the `DECLARE_FEVAL_TABLE` macro, and end the table with the `END_FEVAL_TABLE` macro. Pass the `func1` function pointer to the `FEVAL_ENTRY` macro. The macros perform all the tasks required.

The full form of the macro is:

```
DECLARE_FEVAL_TABLE
    FEVAL_ENTRY(function_name1)
    FEVAL_ENTRY(function_name2)
    (any number of these entries...)
END_FEVAL_TABLE
```

The macros are placed outside of all function definitions and appear after a declaration of the functions being registered.

For example,

```
mwArray function1(mwArray *out, mwArray x, mwArray y);
mwArray function2(mwArray x);

DECLARE_FEVAL_TABLE
    FEVAL_ENTRY(function1)
    FEVAL_ENTRY(function2)
END_FEVAL_TABLE

myfunction()
{
    mwArray a = feval(&b, "function1", c, d);
```

```
    mwArray f = feval("function2", g);
}
```

However, you do not have to register a function in the same file as the call to the function-function or `feval()`. Only one set of macros can appear in any given source file, though you can register additional functions by using the macros in another source file.

4 Call `fmins()` from the main program, passing the string `"func1"` as the first argument, and print the result. `fmins()` computes a local minimizer of `func1()` near its second argument, the scalar `0.25`.

feval( ) Without the Macros

The example is divided into three parts. The first part defines the function `func1()` and shows the main program. The second part specifies the local `feval` function table. The third part defines the thunk function. In the C++ source file, the parts would be combined in this order: `func1()`, the thunk function, the `feval` table code, and `main()`.

```
  // ex3.cpp

  #include <stdlib.h>
① #include "matlab.hpp"

② mwArray func1(mwArray x)
  {
      // One argument test function.
③     return (times(realsqrt(x),reallog(x)));
  }

  int main(void)
  {
④     cout << fmins("func1",0.25) << endl;
      return (EXIT_SUCCESS);
  }
```

The numbered items in the list below correspond to the numbered sections of code example:

1 Include header files. `matlab.hpp` declares the MATLAB C++ Math Library's data types and functions. `matlab.hpp` includes `iostream.h`, which declares the input and output streams `cin` and `cout`. `stdlib.h` contains the definition of `EXIT_SUCCESS`.

2 Declare `func1()`. The name of this function is later passed to `fmins()`. During the execution of `fmins()`, control passes into the MATLAB C Math Library, which calls `func1()`.

3 Compute the natural logarithms and the square roots of the elements in the input matrix and multiply them together. The two functions, `reallog()` and `realsqrt()`, guarantee that their outputs are noncomplex matrices, i.e., matrices that have only real (no imaginary component) elements. Note that this computation means nothing mathematically.

4 Call `fmins()` from the main program, passing the string `"func1"` as the first argument, and print the result. `fmins()` computes a local minimizer of `func1()` near its second argument, the scalar `0.25`.

```
    // This table maps string function names to function pointers. The
    // entries in the table are triplets:
    //
    // <string name> <user function> <thunk function>
    //
    // Every function that can be called by feval() (directly
    // or indirectly) must have an entry in a table like this.

(1) static mlfFuncTabEnt MFuncTab[] =
    {
(2)     { "func1", (mlfFuncp) func1, one_input_one_output },
(3)     { 0, 0, 0 }
    };

    // The following code is a static initializer used
    // to initalize the feval function table. It is intentionally
    // outside the body of any function.

(4) class feval_init {
(5)     feval_init() { mlfFevalTableSetup( MFuncTab ); }
(6)     static feval_init feval_setup;
    };

(7) feval_init feval_init::feval_setup;
```

The numbered items in the list below correspond to the numbered sections of code example:

1 Declare a static global variable, `MFuncTab[]` of type `mlfFuncTabEnt`. This local function table stores one or more function table entries that identify any functions (that you've written) to be executed by a MATLAB C Math Library function-function. In this example, the table stores one entry that identifies the function that `fmins()` executes, `func1()`.

2 Add an entry to the function table. The entry is composed of three parts: a string, `"func1"`, that names the function, a pointer, `(mlfFuncp)func1`, to the function itself, and a pointer, `one_input_one_output`, to the thunk function that actually calls `func1()`.

Notice the `mlf` prefix in the names of the `mlfFuncTabEnt` and `mlfFuncp` types. These are types used by the MATLAB C Math Library and are used to tell the C Math Library function `mlfFeval()` about your function. For more information on the `mlfFuncTabEnt` and `mlfFuncp` types, see the file `matlab.h` in the `include` directory of your MATLAB installation.

3 Terminate the table with a `{0, 0, 0}` entry.

4 Define a private C++ class called `feval_init` that will initialize the local function table.

5 Define a constructor `feval_init()`. In the body of the constructor, pass your function table, `MFuncTab`, to the *C* function `mlfFevalTableSetup()`. Here is another place where the C Math Library interface is used within your C++ application.

6 Declare a class variable named `feval_setup` of type `feval_init`. The class `feval_init` thus contains a static instance of itself. When you define static member data for a class, you must subsequently declare that variable in your code.

7 Now define the variable named `feval_setup` of type `feval_init`. The syntax `feval_init::feval_setup` specifies that the variable is contained within the class `feval_init`. This statement is executed when static variables are initialized. Because `mlfFevalTableSetup()` is called at this time by the constructor, you don't need to explicitly add your entries to the built-in function table maintained by the MATLAB C Math Library.

```
1   typedef mwArray (*PFCN_1_1)(mwArray);

    // This is a "thunk function."
    // The thunk function serves as an interpreter between the
    // MATLAB C++ Math Library's internal feval() mechanism and
    // the user functions.
    // There must be one thunk function for every possible
    // combination of input and output arguments.

2   extern "C" {
3   static
    int one_input_one_output(mlfFuncp pFunc, int nlhs,
                             mxArray **lhs, int nrhs,
4                            mxArray **rhs)
    {
        mwArray Out;

        if ( nlhs > 1 || nrhs > 1 )
5       {
            return(0);
        }

        mwArray tmp = mwArray( rhs[0], 0 );
6
        Out = (*((PFCN_1_1)pFunc))( nrhs > 0 ? tmp
7                                             : mwArray::DIN);

        if (nlhs > 0)
        {
            lhs[0] = Out.FreezeData();
8       }

        return(1);
9   }
    }
```

The numbered items in the list below correspond to the numbered sections of code example:

1 Define the type for the functions handled by a thunk function. The function pointer type that you define here must precisely specify the return type and argument types required by `func1()`.

The `typedef` statement defines a function pointer type, `PFCN_1_1`, that takes one `mwArray` argument and returns an `mwArray`. The name `PFCN_1_1` makes it easy to identify that the function has 1 output argument (the return) and 1 input argument. Use a similar naming scheme when you write other thunk functions that require different numbers of arguments. For example, use `PFCN_2_3` to identify a function that has two output arguments and three input arguments.

2 Declare your thunk function as `extern "C"` to avoid C++ name translation.

3 Declare your thunk function as static to avoid conflicts with other `feval()` calls from other files. Note that if your application requires you to write several thunk functions, and if several of your functions are associated with each thunk function, you may want to group the thunk functions in a separate file. In that case, do not declare the thunk functions static.

4 Define the C-style thunk function that executes `func1()`. A thunk function is a translator between the interface required by the MATLAB C Math Library and your function's interface. You must use the MATLAB C Math Library's `mlfFeval()` calling convention for your thunk function because `mlfFeval()` calls your thunk function from within the C Math Library. Notice the arguments are of type `mxArray` rather than `mwArray`.

The function takes five arguments that describe any one input, one output function (in this example the function is always `func1()`): an `mlfFuncp` pointer that points to `func1()`, an integer (`nlhs`) that indicates the number of output arguments required by `func1()`, an array of `mxArray`'s (`lhs`) that stores the results from `func1`, an integer (`nrhs`) that indicates the number of input arguments required by `func1()`, and an array of `mxArrays` (`rhs`) that stores the input values. `lhs` stands for the left-hand side; `rhs` stands for the right-hand side.

5 Verify that the expected number of input and output arguments have been passed. `func1()` expects one input argument and one output argument. (The return value counts as the one output argument.) Exit the thunk function if too many input or output arguments have been provided.

6 The constructor that builds an mwArray object from an mxArray* has an optional second argument. If this argument is 1 (the default), the mwArray destructor frees the mxArray when its reference count reaches zero. If this argument is 0, as it is in this example, the mwArray destructor will never free the mxArray.

This feature allows you to convert an mxArray to an mwArray temporarily without having the mwArray object free the mxArray when you don't want it to. Use this feature with caution, however, because it can lead to memory leaks; the program must free that mxArray eventually.

7 Call func1(), casting pFunc, which points to func1(), to the type PFCN_1_1. Note that you *must* cast the pointer to func1() to the function pointer type that you defined.

Verify that the expected input argument is provided. If at least one argument is passed to the thunk function, construct an mwArray from the first element in the array of input values (rhs[0]); pass that mwArray, not the mxArray, as the input argument to func1(). Otherwise, pass the special matrix, mwArray::DIN, that MATLAB C++ Math Library functions use to determine the number of inputs. The return from func1() is stored temporarily in the local variable Out, which is already a C++ mwArray.

This line also demonstrates that you can call C++ routines from a C-style function like one_input_one_output(). Be very careful when calling C++ routines from a C routine. You must first manually convert the mxArray arguments into mwArray objects as demonstrated in Note 6. If you do not convert them manually, C++ will do so automatically, with unwanted consequences. The default mxArray to mwArray conversion routine assumes that the mxArray is freed when the last mwArray that references it goes out of scope. This is incorrect for matrices passed to C-style functions like one_input_one_output(). Failure to convert the matrices manually will lead to memory-related bugs that are often hard to track down.

8 Extract the mxArray from the mwArray Out returned by func1(), and assign it to the appropriate position in the array of output values. The return value

is always stored in the first position, `lhs[0]`. If there were additional output arguments, values would be returned in `lhs[1]`, `lhs[2]`, and so on.

The thunk function calling convention requires that a C-style `mxArray` be returned rather than a C++-style `mwArray` object. It is necessary to modify the `mwArray`, `Out`, so that it does not free the `mxArray` it contains when the function terminates and `Out` goes out of scope.

Use the `mwArray` member function `FreezeData()` to modify the `mwArray`. Use it very carefully. `FreezeData()` violates two of the principal design guidelines of the MATLAB C++ Math Library: it reaches into and modifies the array upon which it is invoked, and it provides a mechanism to circumvent the library's automatic memory management. Its effect is to release the `mxArray*` contained by the `mwArray` from automatic memory management.

`FreezeData()` only works on `mwArray` objects that reference `mxArray*`s that have a reference count of one. See "The Space-Time Continuum" in Chapter 7 for more details on reference counting.

9 Return success. A return value of 1 indicates success; 0 indicates failure.

Output

The program produces this output:

```
[
  0.13535
]
```

Representing Input Arguments As a Cell Array

In MATLAB you can substitute a cell array for a comma-separated list of MATLAB variables when you pass input arguments to a function. MATLAB treats the contents of each cell as a separate input argument. To trigger this functionality, you specify multiple values by indexing into the cell array with, for example, the colon index or a vector index.

For example, the MATLAB expression

```
T{1:5}
```

when passed as an input argument is equivalent to a comma-separated list of the contents of the first five cells of `T`. Simply passing the cell array `T` produces an error.

The MATLAB C++ Math Library also supports the expansion of the contents of a cell array into separate input arguments for library functions. For functions that implement MATLAB `varargin` functions, you use `mwArray::cell` to obtain an array reference that returns multiple values.

For example, given the `varargin` function

```
void varargin_func(mwArray a, mwArray b,
                   const mwVarargin &varargin
                   const mwArray v1=mwArray::DIN, ...,
                  .
                  .
                  .
                   const mwArray &v32=mwArray::DIN);
```

you can make the following call:

```
varargin_func(A, B, C.cell(colon(1,5)));
```

`A` and `B`, existing `mwArrays`, are passed as explicit arguments. `C` is a cell array that contains at least five cells. The embedded call to `cell()` uses the index `{1:5}` to return multiple values: the first five cells of C. The MATLAB C++ Math Library passes these as individual arguments to `varargin_func()`.

Location of the Indexed Cell Array in the Argument List

- Pass the return from the cell array indexing operation as one of the variable-length arguments in the input argument list. That reference identifies multiple arrays.
- Do not pass the return from a cell array indexing operation as an explicit argument.

  For example, you cannot make this call to the example `varargin` function.

  ```
  varargin_func(C.cell(1,5), A, B);
  ```

  Given the definition of `varargin_func()`, the first argument position is reserved for an explicit, single argument. The MATLAB C++ Math Library does not handle multiple values in an explicit position.
- You can pass other array arguments or other cell array indexing expressions before or after a cell array indexing expression, all in a `varargin` argument position.

See “Indexing into Cell Arrays” in Chapter 4 to learn more about indexing into cell arrays.

6

Using the Mathematical Operators

Overview

MATLAB supports two types of mathematical operators: *array* operators that operate on individual elements of a matrix and *matrix* operators that operate on whole matrices. In MATLAB, array operators begin with a . (period). Matrix operators do not. For example, .* is the array multiplication operator and * is the matrix multiplication operator.

Array operators treat the elements of each operand individually. Given two operands A and B, an array operator op computes a result C, such that C(i,j) = A(i,j) op B(i,j). The matrices A, B, and C are all the same size.

Matrix operators perform more complex computations. Often the value of an element C(i,j) in the result depends on the values of multiple elements in each input matrix. No single rule describes the relationship between input and output elements for matrix operators. For example, in a matrix multiplication such as C = A * B, the value C(i,j) depends on all of the values in row i of matrix A and column j of matrix B.

This MATLAB code demonstrates the difference between array and matrix multiplication. Note that this is *not* C++ code.

First, initialize two matrices:

```
A = [ 1 2 ; 3 4 ];
B = [ 1 0 ; 0 1 ];   % Identity matrix
```

Now compute the array product (array multiplication):

```
C = A .* B
C =
        1 0
        0 4
```

Now compute the matrix product (matrix multiplication):

```
D = A * B
D =
        1 2
        3 4
```

After this MATLAB code is executed, the matrix C contains the array product [1 0; 0 4]. Since C was computed by array multiplication, the elements of C equal:

```
C(1,1) = A(1,1) * B(1,1)
C(1,2) = A(1,2) * B(1,2)
C(2,1) = A(2,1) * B(2,1)
C(2,2) = A(2,2) * B(2,2)
```

The matrix D, on the other hand, contains the linear-algebraic product of A with the identity matrix: A itself. The equivalent C++ code is presented at the end of this section on page 6-5.

Note Array operators work with N-dimensional arrays; matrix operators work with two-dimensional array.

Using the Operators

Many of MATLAB's mathematical operators (+, –, *, /, ^) are the same as those available in C++. The exceptions are ', \, and the array operators .*, ./, .\, and .^, because the syntax of C++ does not support their definition as operators. You must use the functional equivalents provided by the MATLAB C++ Math Library to perform these operations.

This table demonstrates how the library supports mathematical operators. Note that the library also provides functional equivalents for the set of operators that are supported by C++ syntax.

Table 6-1: MATLAB Operator and C++ Function Equivalence

Description	Definition: C = A <op> B	MATLAB Operator	C++ Operator	C++ Function
Array multiplication	`C[i] = A[i] * B[i]`	`.*`	None	`times()`
Array right division	`C[i] = A[i] / B[i]`	`./`	None	`rdivide()`
Array left division	`C[i] = B[i] / A[i]`	`.\`	None	`ldivide()`
Array exponentiation	`C[i] = A[i] ^ B[i]`	`.^`	None	`power()`
Array addition	`C[i] = A[i] + B[i]`	`+`	`+`	`plus()`
Array subtraction	`C[i] = A[i] – B[i]`	`–`	`–`	`minus()`
Matrix multiplication	Inner product	`*`	`*`	`mtimes()`
Matrix right division	`C` such that `C*B = A`	`/`	`/`	`mrdivide()`
Matrix left division	`C` such that `A*C = B`	`\`	None	`mldivide()`
Matrix exponentiation	`C = A*A*...*A` (`B` times)	`^`	`^`	`mpower()`
Complex transpose	N/A (unary)	`'`	None	`ctranspose()`
Transpose	N/A (unary)	`.'`	None	`transpose()`

With the exception of the unary `transpose()` and `ctranspose()` functions, the C++ functions in the table take two matrix arguments and return a third matrix. To see these functions in action, consider the C++ translation of the MATLAB code presented on page 6-2 at the beginning of this section. Function calls replace the use of operators. (Note that * can be used instead of `mtimes`.)

```
static double data[] = { 1, 3, 2, 4 };
mwArray A(2, 2, data);
mwArray B = eye(2);          // 2x2 identity matrix

mwArray C = mtimes(A, B);    // Matrix multiplication
cout << C << endl;

mwArray D = times(A, B);     // Array multiplication
cout << D << endl;
```

Running this code fragment produces:

```
[
    1 2
    3 4
]
[
    1 0
    0 4
]
```

Use the other binary operator functions in a similar manner.

Defining Your Own Operators

Defining your own operator in C++ is called "overloading an operator." Strictly speaking, you cannot define a new operator; you can only provide an alternative definition for an existing operator. The set of operators that you can overload is limited to the set recognized by C++ but not defined by the MATLAB C++ Math Library.

For example, C++ does not recognize the character sequence ** as an operator. If you try to define `operator**()` to mean exponentiation, the compiler will issue a syntax error. However, you can define a matrix equivalent for any recognized operator that is missing from the library. You define it in terms of the operators that do come with the library. See Chapter 11 for a complete list of the operators.

Defining matrix equivalents for the additional operators that C++ defines is a simple process. The following example illustrates the proper way to define a new operator.

Assume that you want to define `operator*=()`, which combines the multiplication and assignment operations. Because the library predefines `operator*()` and `operator=()`, building `operator*=()` is straightforward.

```
mwArray operator*=(mwArray &A, const mwArray &B)
{
    A = A * B;
    return A;
}
```

The above code overloads `operator*=()` for matrix arguments. It is important, in this case, to return the modified matrix, so that you can concatenate the operator with other operators, for example, `C = A *= B;`. Although the coding style of this example is poor, the code is legal.

When you overload an operator in C++, you cannot change the arity (number of operands) or precedence of the operator. For example, the C++ language definition restricts `operator+()` to two arguments. You cannot define an `operator+()` that takes three arguments and returns the sum of all three. Similarly, you cannot change the precedence of `operator+()` to make the addition in the expression `a+b*c` occur before the multiplication. Use parentheses to change operator precedence on an expression-by-expression basis. For more information on overloading operators, consult a C++ reference guide.

7

Printing, Exceptions, and Memory Management

This chapter describes how to write print handlers, handle exceptions, and replace default memory management.

Defining a Print Handler

The MATLAB C++ Math Library is designed to run on character-based terminals and in graphical, windowed environments. Simply using `printf()` or a similar routine is fine for character-terminal output but insufficient for output in a graphical environment. To support programs with graphical user interfaces, the library allows you to specify how it displays output.

The MATLAB C++ Math Library performs some output, in particular it displays error messages and warnings, but doesn't perform input. The MATLAB C++ Math Library's output requirements are very simple. The library formats its output into a character string internally and then calls a function, the *print handler*, that prints the string. If you want to change where or how the library's output appears, you must provide an alternate print handler.

Providing Your Own Print Handler

By default, the library sends output to the C++ standard output stream, `cout`. However, instead of sending output directly to the standard output stream, the MATLAB C++ Math Library calls a print handler when it needs to display an error message or warning. The print handler used by the library takes a single argument, a `const char *` (the message to be displayed), and returns `void`.

The default print handler:

```
static void DefaultPrintHandler(const char *s)
{
    cout << s;
}
```

If you want to perform a different style of output, you can write your own print handler and register it with the MATLAB C++ Math Library. Any print handler that you write must match the signature of the default print handler: a single `const char *` argument and a `void` return.

To register your function and change which print handler is used, you must call the routine `mwSetPrintHandler`. `mwSetPrintHandler` takes a single argument, a pointer to a function that displays the character string, and returns `void`.

```
void mwSetPrintHandler(mwOutputFunc f);
```

Using the Print Handler to Print Your Own Messages

The print handler is not reserved for the exclusive use of the MATLAB C++ Math Library. Once you've written a print handler for the library to use, you can also use it to print messages of your own.

You may either call your print handling routine directly, or call the function `mwGetPrintHandler()`, which returns a pointer to the current print handling function. The following example function demonstrates how to call `mwGetPrintHandler()` and what to do with the result.

```
#include "matlab.hpp"

void hello()
{
    mwOutputFunc f = mwGetPrintHandler();

    (*f)("Hello world\n");
}
```

Output to a GUI

The next two sections illustrate how to provide an alternate print handler under the X Window System and Microsoft Windows. When you write a program that runs in a graphical windowed environment, you can display printed messages in informational dialog boxes.

These examples present a simple alternative output mechanism and demonstrate the interface between the MATLAB C++ Math Library and each of the windowing systems. There are other output options as well, for example, sending output to a window or portion of a window inside an application. The code in these examples should serve as a solid foundation for writing more complex output routines.

The examples assume that you know how to write a program for a particular windowing system and, therefore, omit code that is common to such programs, for example, the application start-up and initialization code is missing. Please consult your windowing system's documentation if you need more information than the examples provide.

Note If you use an alternate print handler, you must call `mwSetPrintHandler()` before calling other library routines. Otherwise the library uses the default print handler to display messages.

X Windows System/Motif Example

The Motif Library provides a `MessageDialog` widget that this example uses to display text messages. The `MessageDialog` widget consists of a message text area placed above a row of three buttons: **OK**, **Cancel**, and **Help**.

The `MessageDialog` box is a modal dialog box; while it is posted, this application will not accept input. You must press the **OK** button to dismiss the `MessageDialog` dialog box before you can do anything else. However, since the `MessageDialog` is a child of the application, and not the root window, other applications will continue to operate normally.

```
/* X-Windows/Motif Example */

/* List other X include files here */
#include <Xm/Xm.h>
#include <Xm/X11.h>
#include <Xm/MessageB.h>

static Widget message_dialog = 0;

/* The alternate print handler */
void PopupMessageBox(const char *message)
{
    Arg args[1];

    XtSetArg(args[0], XmNmessageString, message);
    XtSetValues(message_dialog, args, 1);
    XtPopup(message_dialog, XtGrabExclusive);
}

main()
{
    /* Start X application. Insert your own code here. */
    main_window = XtAppInitialize( /* your code */ );
```

```
    /* Create the message box widget as a child of */
    /* the main application window. */
    message_dialog = XmCreateMessageDialog(main_window,
                        "MATLAB Message", 0, 0);

    /* Set the print handler */
    mwSetPrintHandler(PopupMessageBox);

    /* The rest of the program */
}
```

This example declares two functions: `PopupMessageBox()` and `main()`. `PopupMessageBox` is the print handler and is called every time the library needs to display a text message. It places the message text into the `MessageDialog` widget and makes the dialog box visible.

The second routine, `main()`, first creates and initializes the X Window system application. That code is not shown but can be found in an X Windows reference guide. `main()` then creates the `MessageDialog` object used by the print handling routine. Finally, `main()` calls `mwSetPrintHandler()` to make the library call `PopupMessageBox()` instead of the default print handler. If this were a real application, the main routine would continue with calls to other routines or code to perform computations.

Microsoft Windows Example

This example uses the Microsoft Windows `MessageBox` dialog box. This dialog box contains an "information" icon, the message text, and a single **OK** button. The `MessageBox` is a Windows modal dialog box; while it is posted, no other application will accept input. You must press the **OK** button to dismiss the `MessageBox` dialog box before you can do anything else.

This example declares two functions. The first, `PopupMessageBox()`, places the message into the message box and then posts the box to the screen. The second, `main()`, creates and starts the Windows application (that code is not shown), and then calls `mwSetPrintHandler()` to set the print handling routine to `PopupMessageBox()`.

```
/* Microsoft Windows example */

static HWND window;
```

```
static LPCSTR title = "Message from MATLAB";

/* The alternate print handler */
void PopupMessageBox(const char *message)
{
    MessageBox(window, (LPCTSTR)message, title,
                MB_ICONINFORMATION);
}

main()
{
    /* Register window class and provide window procedure. */
    /* Fill in your own code here. */

    /* Create application main window. */
    window = CreateWindowEx( /* Whatever */ );

    /* Set print handler. */
    mwSetPrintHandler(PopupMessageBox);

    /* The rest of the program ... */
}
```

This example does no real processing. If it were a real program, the main routine would contain calls to other routines or perform computations of its own.

Handling Exceptions

The MATLAB C++ Math Library delivers error messages via exceptions. This section:

- Provides a brief overview of C++ exception-handling
- Describes how to handle C++ Math Library exceptions in your application
- Describes how you can customize exception handling by replace the default exception handling routines with routines of your own design.
- Describes how the MATLAB C++ Math Library implements exceptions, and how you can use this mechanism to throw and catch exceptions in your own code.

Refer to Appendix C for a list of MATLAB C++ error messages.

C++ Exception Handling Overview

Many earlier error-handling schemes reported errors via a return value from a function. That mechanism was inconvenient and unreliable for two reasons. First, it did not allow function composition, where one function call is nested in the argument list of another. For example, `f(g(x))` composes `f()` and `g()`. Second, the scheme placed the burden for checking error codes on the programmer.

Many other schemes, including the one used by the standard C library, use a global variable in place of returned error codes. This mechanism solves one problem, function composition, but still requires that the programmer check for errors.

The C++ exception-handling mechanism suffers from neither of these problems. Exception-handling does not require that each function return an error code, which means that functions can be composed. In addition, exceptions cannot be ignored by a programmer because an uncaught exception terminates the program. If a programmer forgets to handle an exception, abrupt program termination is a potent reminder.

Handling C++ Math Library Exceptions in Your Code

Your programs *must* catch exceptions thrown by the MATLAB C++ Math Library. Uncaught exceptions cause abnormal program termination.

To handle these exceptions, you need to catch each exception and display the message associated with it. C++ provides the mechanism for catching exceptions: the `try` and `catch` keywords.

Here's a basic example that demonstrates these techniques.

```
// try-block
try
{
     eig(A);
}

//catch-block
catch(mwException &ex)
{
    mwDisplayException(ex);
}
```

The `try` keyword introduces a try block. Any exception thrown while executing the code in the try block transfers control to the first catch block that applies to the type of the exception being caught. Each catch block catches one type of exception. You use multiple catch blocks to catch exceptions of different types.

This catch block catches any exception objects derived from the class `mwException`. `mwException` is the superclass for all exceptions thrown by the MATLAB C++ Math Library. If you catch and display exceptions of this type, you see all the error messages associated with the exceptions thrown by the library. For a complete list of the exceptions defined by the MATLAB C++ Math Library, see Appendix B.

Example Program: Handling Exceptions (ex5.cpp)

This example demonstrates the MATLAB C++ Math Library's error handling facilities. The program deliberately triggers a library exception by specifying a negative number as an array index. You can find the code for this example in the `<matlab>/extern/examples/cppmath` directory on UNIX systems and in the `<matlab>\extern\examples\cppmath` directory on PCs, where `<matlab>` represents the top-level directory of your installation. See "Building C++ Applications" on page 1-13 for information about building and running the example program.

In the example, note the following:

- You should always have a try and catch block in your main routine.
- Exceptions are caught by object type. `mwException` is the top-level exception class.
- A C++ program may have multiple catch blocks, each of which catches different types of exceptions.
- Once an exception is thrown, it propagates up the call stack until it reaches the first catch block that catches exceptions of its type.
- All the exceptions in the MATLAB C++ Math Library contain an associated error message.

Note This example uses a simple exception handling mechanism and does not use nested try blocks or multiple catch blocks. Though these C++ features are compatible with the MATLAB C++ Math Library, they are beyond the scope of this book. Refer to your C++ reference manual for information on nested try blocks and multiple catch blocks.

```
    // ex5.cpp

    #include <stdlib.h>
    #include "matlab.hpp"

    static double data[] = { 1, 2, 3, 4, 5, 6 };

    mwArray compute(const mwArray &in)
    {
        // Cause an error: use a negative index.
1       return in(-5) * 17; // <1>
    }

    int main(void)
    {
        // Handle exceptions for all code in the try-block.
2       try { // <2>
            mwArray mat0(2, 3, data);
            mwArray mat1;

3           mat1 = compute(mat0); // <3>
            cout << mat1 << endl;
        }
        // Catch and print any exceptions that occur.
4       catch (mwException &ex) { // <4>
            cout << ex << endl;
            return(EXIT_FAILURE);
        }
        return(EXIT_SUCCESS);
    }
```

The numbered items in the list below correspond to the numbered sections of code example:

1 Deliberately cause an error. The `compute()` function attempts an illegal operation: a negative number was used as a matrix index. This operation causes the MATLAB C++ Math Library to throw an exception. An exception propagates up the call-chain, or stack, until it reaches a catch block that handles it. Even though the `compute()` function does not contain a catch block, the exception is properly handled by the catch block in `main()`.

2 Begin a try block. The `try` keyword introduces the block. A try block is like a safety net. try blocks are always followed by one or more catch blocks. The catch block that follows a try block processes any exceptions thrown during execution of the try block. In addition to catching exceptions that are generated by the code in the try block, the catch block catches exceptions thrown by the functions called from within the try block and by the functions called from within those functions, and so on.

An exception is a C++ object. Every C++ object has a type. When an exception is thrown, the exception stops at the nearest (in terms of the call chain, or stack) catch block that handles exceptions of its type. In this case, because there is only one catch block in the program, there is only one place for the exceptions to stop.

3 Call the `compute()` function. If the function does not throw an exception, the value of `mat1` is printed. Note the absence of code to test for an error from `compute()`. The lack of error-checking code makes the rest of the code easier to follow. This is another of the advantages of exception handling: catch blocks separate error-handling code from ordinary code, making the rest of the function easier to read and maintain.

4 Begin a catch block. The `catch` keyword introduces the block. Catch blocks are always associated with `try` blocks. Catch blocks "catch" or stop a propagating exception according to the type of the exception.

This catch block catches all exceptions of type `mwException` or one of its subclasses. `mwException` is the base exception class for the MATLAB C++ Math Library. Exceptions not caught by this catch block, i.e., any exceptions not of type `mwException`, cause the abrupt termination of the program. Depending on the operating system, an error message may or may not be printed.

Output

The program produces this output:

```
WARNING: Subscript indices must be integer values.
RuntimeError
Exception!  File: handler.cpp, Line: 169
    Index into matrix is negative or zero.  See release notes on
    changes to logical indices.
```

In general, printing an `mwException` using a C++ output stream produces two output lines. The first describes the type of exception (generic, in this case) and identifies the file and line number where the exception was thrown. The file name and line number often refer to MATLAB C++ Math Library code rather than to your code. They indicate the origin of the exception, rather than where it was caught or where the error occurred in your code.

The second line describes the exception in more detail. In this case, the message tells you that an illegal indexing operation occurred. The subscript, `-5`, was applied to a matrix for which the valid subscripts fall between one and six, inclusive.

Replacing the Default Library Error Handler

The default error handling behavior of the MATLAB C++ Math Library routines is implemented by the default library *error handler* routine. You can customize error handling by replacing the default library error handler routine with one of your own design.

Note If your compiler supports exceptions, the error handler is only called for warning-level errors. For all other errors, a C++ exception is thrown.

To replace the default error handler you must:

- Write an error handler
- Register your error handler so that library routines call it when they encounter an error

Writing an Error Handler

When you write an error handler, you must conform to the library prototype for error handling routines:

```
void MyErrorHandler( const char *msg, bool isError  )
{
    if(isError) // Will always be false if exceptions supported
    {
       // Process Error
    }
     else
```

```
        {
           // Process Warning
        }
    }
```

In this prototype, note the following:

- An error handling routine must not return a value (return void).
- An error handling routine accepts two arguments, a `const` string and a Boolean value. The string is the text of the error message. When the value of this Boolean value is `TRUE`, it indicates an error message. If this value is `FALSE`, it indicates a warning message.

Registering Your Error Handler

After writing an error handler, you must register it with the MATLAB C Math Library so that the library routines can call it when they encounter an error condition at runtime. You register an error handler using the `mwSetErrorMsgHandler()` routine.

```
mwSetErrorMsgHandler(MyErrorHandler);
```

Exception Handling in the MATLAB C++ Math Library

Note You only need to read this section if your C++ compiler does not support exception handling. Check your compiler documentation to see if it includes this support.

Because not all C++ compilers fully support exception handling, the MATLAB C++ Math Library, provides an alternative exception handling mechanism. The `mwException` class defines a virtual function called `do_raise()`. Instead of using the `throw` keyword to throw an exception, the library code calls `do_raise()` instead. When built with a compiler that fully supports exceptions, the `do_raise()` function throws the exception using the `throw` keyword. Otherwise, `do_raise()` prints the exception and calls `exit(-1)`.

The disadvantage to this approach is that in an environment without support for exceptions, all exceptions are automatically fatal. However, compiler support for exceptions is growing more widespread rapidly, so this situation should be temporary.

Using the MLM_THROW Macros to Throw Exceptions

In order to make this dual support transparent, all exceptions are thrown with `do_raise()` rather than `throw`. Six macros make this dual mechanism easy for you to use in your own code.

The macros are named `MLM_THROW<X>`, where `<X>` is an integer from 0 to 5. The integer suffix indicates the number of *additional* arguments that the macro takes. Each macro takes at least two arguments (not counted in the integer suffix): the type of exception to throw and a text string message that describes the problem. The type of exception corresponds to the name of one of the exception classes documented below. Additional arguments are text strings, integers, or doubles that substitute for format specifiers in the first string argument. The number of format specifiers correspond to the number of additional arguments.

The macros process the message and any extra arguments with `sprintf()`. `MLM_THROW3()`, for example, takes five arguments: the type of exception, the text string message, and three additional arguments. This mechanism lets you write descriptive error messages.

The last member of the set of macros is `MLM_THROW5()`. If you need to pass more than five additional arguments to `MLM_THROW<X>`, you must write additional macros. Look in the file `mlmexcpt.h` for the definitions of the macros and pattern your new macros after them. You'll find the header in the `<matlab>/extern/include/cpp` directory, on Unix systems, or the <matlab>\extern\include\cpp directory, on PCs, of your MATLAB C++ Math Library installation.

The following example taken from the MATLAB C++ Math Library's indexing code demonstrates the use of the `MLM_THROW2` macro. The indexing code verifies that an index that accesses array data is valid. If a specified index is less than the minimum, or base, index, the library throws an `mwDomainError` exception.

```
if (i < index_base)
    MLM_THROW2(mwDomainError, \
     "An index (%ld) was less than %ld, the minimum legal index."\
               i, index_base)
```

Two things to note about this code:

- The keyword throw does not appear. The macro itself throws the exception.
- The MLM_THROW2() statement is similar to an ordinary printf() call. MLM_THROW2() passes its second, third, and fourth arguments to sprintf(), which formats them just as printf() would.

Note The backslashes at the ends of lines are required because MLM_THROW2() is a macro rather than a function call. Backslashes would not be necessary if the entire call to MLM_THROW2() fit on a single line.

Including an mwArray in an Exception Message

The subclasses of mwException contain text-based messages describing the error that triggers the exception. In many cases, faulty data causes the problem, in which case it may be useful to include part of the array data in the error message.

The constructors of the exception classes take a string as an argument. Using the standard C++ class strstream, you can produce a string representation of all or part of an mwArray.

```
mwArray A = rand(4);
strstream string;
string << "This matrix: " << A << "caused the problem." << endl
       << ends;
MLM_THROW0(mwRangeError, string.str());
```

This code formats an error message in a strstream, which dynamically grows to accommodate the data stored in it. Calling str() on the strstream freezes it so that the strstream can no longer grow. str() then returns the string stored in the strstream.

Memory Management

The MATLAB C++ Math Library manages memory efficiently by allocating space for new arrays and then freeing the space when the memory is no longer in use. The library, like many C++ components, makes extensive use of temporary variables, many of which are dynamically allocated. The resulting number of allocations and deallocations is too large for the operating system's default memory management to handle with acceptable performance.

To handle this large number of allocations and frees, the library implements its own memory management system that replaces the operating system's default memory management scheme. The MATLAB C++ Math Library avoids an excessive number of calls to `malloc()` and `free()` by maintaining a memory pool of its own. This pool grows to accommodate the memory needs of your program.

Setting Up Your Own Memory Management Routines

Because this default memory management may not be appropriate for all applications, we provide the function `mwSetLibraryAllocFcns()` that you can use to register your own memory management routines:

```
void mwSetLibraryAllocFcns(mwMemCallocFunc callocProc,
                           mwMemFreeFunc freeProc,
                           mwMemReallocFunc reallocProc,
                           mwMemAllocFunc mallocProc);
```

The types defined for the arguments to `mwSetLibraryAllocFcns()` are:

```
typedef void *(*mwMemCallocFunc)(size_t, size_t);
typedef void (*mwMemFreeFunc)(void *);
typedef void *(*mwMemReallocFunc)(void *, size_t);
typedef void *(*mwMemAllocFunc)(size_t);
```

Note The `mwSetLibraryAllocFcns()` routine must be called before any `mwArray` objects are declared, because declaring an `mwArray` object causes memory to be allocated.

To set up your own memory management routines, you need to write four routines: two memory allocation routines, one memory reallocation routine, and one deallocation routine. You then call `mwSetLibraryAllocFcns()` to register those routines with the library.

Note You cannot omit any of the four routines. You must supply them all.

For example, this call registers the standard C++ memory management routines with the MATLAB C++ Math Library. (Note, however, that using the standard C++ memory management routines will decrease the performance of the MATLAB C++ Math Library.)

```
mwSetLibraryAllocFcns(calloc, free, realloc, malloc);
```

Note Do not call the MATLAB C Math Library function `mlfSetLibraryAllocFcns()` from your application.

Calloc Allocation Routine

Any memory calloc routine that you write must conform to the type:

```
typedef void *(*mwMemCallocFunc)(size_t, size_t);
```

The calloc function allocates a block of memory based on the number of contiguous elements that you want allocated (its first argument) and an integer representing the size of each element (its second argument). The routine initializes the allocated memory to zero.

```
static void *SampleUserCalloc(size_t count, size_t size)
{
    // function body
}
```

Deallocation Routine

If you write a memory allocation routine, you must write a corresponding routine that frees memory. Any memory free routine that you write must conform to this type:

```
typedef void (*mwMemFreeFunc)(void *);
```

The free function takes a pointer to the beginning of the memory block to be freed and returns void.

```
static void SampleUserFree(void *ptr)
{
    // function body
}
```

The overloaded `delete` operator in `mwArray` calls this function, as does `mxFree()`.

Reallocation Routine

Any memory reallocation routine that you write must conform to this type:

```
typedef void *(*mwMemReallocFunc)(void *, size_t);
```

The realloc function takes a pointer to the beginning of the memory block to reallocate and an integer size of each element. It returns a pointer to void.

```
static void *SampleUserRealloc(void *ptr, size_t size)
{
    // function body
}
```

Malloc Allocation Routine

Any memory allocation routine that you write must conform to this type:

```
typedef void *(*mwMemAllocFunc)(size_t);
```

The malloc function takes an integer size that represents the number of bytes to allocate and returns a pointer to void. Unlike calloc, malloc does not initialize the memory it returns.

```
static void *SampleUserMalloc(size_t size)
{
    // function body
}
```

The overloaded `new` operator in `mwArray` calls this function, as do the `mx`-prefixed allocation routines, for example, `mxMalloc()`.

Performance and Efficiency

You do not need to understand the information in this section to use the MATLAB C++ Math Library effectively. It is included to satisfy the curious and provide a glimpse into the inner workings of the library. Reading this section may enable you to eke those last few microseconds out of a tight loop or decrease your program's memory requirements, but be warned that the information presented here is subject to change without notice.

In general, performance and efficiency are tightly linked to implementation. Should the implementation of the MATLAB C++ Math Library change (as it is likely to), the most efficient way to use the library will likely change as well. This is a warning. If you take advantage of the descriptions below to increase the speed of your code, be aware that the next release of the library may do things differently, and your highly tuned code may run more slowly than you expect.

The Space-Time Continuum

Faster, smaller, cheaper: choose any two. It is well-known that programs can be made more space efficient at the cost of decreasing their time efficiency, and vice versa. The code in the C++ library makes trade-offs, as described below, in an attempt to execute as rapidly as possible, without using excessive amounts of memory.

Time

The MATLAB C++ Math Library is implemented on top of the MATLAB C Math Library, which in turn is a layer above the raw MATLAB code. Despite this layering, the C++ library code performs well. The intermediate layers consume less than 1% of a typical program's CPU time.

During the development of the C++ library, one of the greatest increases in speed resulted from the implementation of a block-caching memory manager. This is a classic example of trading space for time. The space cost of maintaining an internal list of memory blocks eliminates the time cost of a system call to `malloc()`. This time savings can be quite significant. On the PC, for example, this system resulted in a seven-fold increase in speed.

Space

There are two major motivations for a space-efficient implementation. The first motivation is the obvious one: the more arrays that you create or the larger arrays that you create, the more interesting problems you can solve. The second motivation is less obvious but equally important: allocating blocks of memory is slow and, thus, the fewer allocated, the better the program's performance.

C++ is notorious for copying objects and automatically creating and destroying many temporary objects. This behavior is particularly common in arithmetic expressions and in passing arguments to functions. Since these temporary objects cannot be avoided, it is very important that they be inexpensive, both in terms of time and space.

The current implementation uses a reference-counting scheme to minimize the size of a copy. Using this scheme, each copy requires at most an additional eight bytes, regardless of the size of the copied array. Aside from allocating the space, no additional computation is necessary to make the copy. This representation is quite efficient. Since MATLAB functions and operations have no side effects, and the assignment operator practices copy-on-write, reference counting is safe.

8

Array Input and Output

Overview

This chapter describes how to:

- Use C++ input and output streams to read and write from input and output from files.
- Use `fscanf()` and `fprintf()` to read and write from input and output from files.
- Read data from and write data to MAT-files

The C++ I/O stream operators are more convenient than functions like `fprintf()`, because they are more consistent, flexible, and extensible. Because each object in a C++ program is responsible for printing itself to a stream and reading itself from a stream, objects have complete control over their own printed format. New objects can be added without changing the code in the basic streams mechanism.

In addition, the MATLAB C++ Math Library also supports the `load()` and `save()` routines which enable you to import and export data in MAT-file format. The library stream I/O implementations do not support .

Using Array Stream I/O

This section:

- Provides an overview of C++ stream I/O
- Details how to form a stream I/O format specification
- Describes how to use stream I/O with files
- Describes how to use stream I/O to perform interprocess communication

This section includes a complete example program.

Overview

A C++ stream is a sequence of data objects. Often a stream consists of a sequence of characters. C++ streams encompass all the I/O devices attached to a computer (keyboard, screen, disk, etc.).

There are two basic types of streams: input streams and output streams. Streams can be attached to one of many types of data sources, or sinks, such as files, strings, and the screen, so that input can be read from and written to both disk files and the user's terminal. Refer to your C++ reference for a complete explanation of streams and C++'s input and output facilities.

C++ defines three standard streams, `cin`, `cout`, and `cerr`. `cin` is bound to standard input, `cout` to standard output, and `cerr` to standard error. The MATLAB C++ Math Library provides standard C++-style stream input (>>) and output (<<) operators for `mwArray` objects. For example, to send an array `A` to the standard output, you write:

```
cout << A << endl;
```

To read an array in from standard input, you write:

```
cin >> A;
```

To send an array `A` to standard error, you write:

```
cerr << A << endl;
```

Note The MATLAB C++ Math Library supports stream *input* and *output* of multidimensional numeric arrays. However, the library only supports stream *output* of cell arrays, sparse matrices, and structures. These arrays may be printed using the << operator but they are not printed in a format which can be read back in using the >> operator.

Example – Array Stream I/O (ex1.cpp)

This example illustrates the use of stream I/O to read in and print out a MATLAB array. You can find the code for this example in the `<matlab>/extern/examples/cppmath` directory on UNIX systems and in the `<matlab>\extern\examples\cppmath` directory on PCs, where `<matlab>` represents the top-level directory of your installation. See “Building C++ Applications” on page 1-13for information about building and running the example program.

In the example, note that the array input format is the same as the array output format. Data written out by a program can be easily read back in by a program. For more information about this format, see the “Stream I/O Format Definitions” on page 8-7.

```
    // ex1.cpp

    #include <stdlib.h>
(1) #include "matlab.hpp"

    static double data[] = { 1, 2, 3, 4, 5, 6 };

    int main(void)
    {
        // Create two matrices.
        mwArray mat0(2, 3, data);
        mwArray mat1(3, 2, data);

        // Print the matrices.
(2)     cout << mat0 << endl;
        cout << mat1 << endl;

        // Read a matrix from standard in, then print the matrix to
        // standard out.
(3)     cout << "Please enter a matrix: " << endl;
        cin >> mat1;
(4)     cout << mat1 << endl; // ** See Note 4 **

        return (EXIT_SUCCESS);
    }
```

The numbered items in the list below correspond to the numbered sections of code example:

1 Include header files. `matlab.hpp` declares the MATLAB C++ Math Library's data types and functions. `matlab.hpp` includes `iostream.h`, which declares the input and output streams `cin` and `cout`. `stdlib.h` contains the definition of `EXIT_SUCCESS`.

2 Print the matrices using the C++ standard output stream, `cout`. By default, objects printed with `cout` appear on the screen, though you can redirect the output to a file.

3 Prompt the user to type in a matrix. Read the matrix into `mat1` using the C++ standard input stream, `cin`. The matrix does not need to be the same size as the matrix already stored in `mat1`. The input operator >> creates a new matrix and assigns that matrix to `mat1`. UNIX and PC systems read from the terminal by default; you can redirect them to read from an input file.

4 Print the newly read matrix.

Output

The program prints out the two matrices, `mat0` and `mat1`, and then prompts the user to input an array.

```
[
        1         3         5 ;
        2         4         6
]

[
        1         4 ;
        2         5 ;
        3         6
]

 Please enter a matrix:
```

To enter a matrix, first type in a left bracket ([) character, and then enter a series of numbers. You can insert semicolons at any point to create a two-dimensional matrix. End your matrix by typing a right bracket (]) character. Each row must contain the same number of columns. Spaces, tabs, and carriage returns are ignored. For complete information about input and output formats, see the section “Stream I/O Format Definitions” on page 8-7.

For example, if you type in `[ 1 2; 3 4 ]`, the program prints it.

```
[
        1      2;
        3      4
]
```

Note that the output format is the same as the input format, enabling the output from one program to be used as the input to another. Because each matrix is delimited by [and], input files or streams can contain more than one matrix.

Stream I/O Format Definitions

The MATLAB C++ Math Library input and output formats strongly resemble their interpreted MATLAB counterparts. The array output format conforms to the rules for array input, which means that arrays written to a stream using << can be read in from a stream using >>.

Note The >> and << operator implementations do not read and write MAT-files. Use the functions `load()` and `save()` to read and write MAT-files. See the section "Importing and Exporting MAT-File Data" on page 8-20 for more information.

In the MATLAB C++ Math Library, special input characters describe the shape of the array. The `[` and `]` characters (brackets) enclose an array definition. The { and } characters (braces) enclose a cell array definition. Within the brackets or braces, the contents of the array appear in row-major order. A semicolon (`;`) separates rows.

The following table lists the syntax elements in the format definition.

Note The >> operator implementation cannot read cell arrays, sparse arrays, and structures.

Table 8-1: Elements of mwArray Input/Output Syntax

Syntax Element	Definition	Example
`[]`	Encloses array definition	`[ 1 2 ]`
`{}`	Encloses a cell array definition	`{[1 2] 'Eric'}`
`e`	Indicates scientific notation	`1e7`

Table 8-1: Elements of mwArray Input/Output Syntax (Continued)

Syntax Element	Definition	Example
`.`	Indicates floating-point number	`1.879`
`–`	Indicates negative number or exponent	`–1.3e–8`
`+`	Separates complex and imaginary parts	`1+2i`
`i`	Indicates complex number	`1+2i`
`'`	Encloses a string	`'abcd'`
`.`*fieldname*	Identifies a field in a structure	
`;`	Separates rows	`[ 1 2 ; 3 4 ]`
`*`	Separates optional scaling factor from array	`1e–10 * [ 1 2 ]`
whitespace	Separates array elements	`[ 1 2 ]`

Legal Array Elements

- Integers
- Floating-point numbers
- Complex numbers
- Strings

You can also specify a scaling factor that modifies the values in an `mwArray` containing integers, floating-point numbers, or complex numbers. A scaling factor applies equally to all elements in the array and is used to enter very small or very large values.

Length of Input Array

The only restriction on the length of the input array is the amount of memory available; the input mechanism imposes no restrictions of its own.

How Whitespace Is Interpreted

- The input operator ignores additional space and tab characters between array elements.
- The input operator ignores whitespace between array definitions.

Characteristics of Input Files

Input files must be in ASCII rather than binary format. An input file may contain multiple array definitions. Whitespace between array definitions is ignored.

Differences between MATLAB and the C++ Math Library

There are differences between the input accepted by MATLAB and the MATLAB C++ Math Library. MATLAB input files permit the use of MATLAB mathematical expressions in array definitions. The MATLAB C++ Math Library does not support the use of functions or operators in input streams.

For example, the MATLAB C++ Math Library does *not* support this:

```
[ (1 + 2) 7; 4 5]
```

MATLAB does.

Specifying an Array for Input

1 Begin the array with a left bracket [.

2 List the elements in the first row of the array in row-major order.

3 Separate the first row from the second row with a semicolon (;).

4 Repeat steps 2 and 3 for each row in your array.

5 Close the array with a right bracket]. However, do not end the last row with a semicolon.

Table 8-2: Input Syntax for an mwArray Containing Integers

Input	Array	Array Type
`[ 1 2 ; 3 4 ]`	`1 2` `3 4`	2-by-2 square array

`[ 1 2;`
`3 4;`
`5 6 ]`
	`1 2`
	`3 4`
	`5 6`
		3-by-2 rectangular array

Table 8-3: Input Syntax for an mwArray Containing Floating Point Numbers

Input	Array	Array Type
`[ 1.4  2.5  3.2 ]`	`1.4  2.5  3.2`	1-by-3 vector

`[`
`3.14e2  2.73e4;`
`1.73e3  1.41e2`
`]`
	`314    27300`
	`1730     141`
		2-by-2 square array. Note use of scientific notation in input.

Table 8-4: Input Syntax for an mwArray Containing Complex Numbers

Input	Array	Array Type

`[`
`   1+3i  2+7i ;`
`   9–5i  8+4i`
`]`
	`1+3i  2+7i`
	`9–5i  8+4i`
		2-by-2 complex square array

Table 8-5: Input Syntax for an mwArray Containing Strings

Input	Array	Array Type
`'abcd'`	`abcd`	1-by-4 character array Equivalent to `['abcd']`

`[`
` 'abcd';`
` 'efgh';`
`]`
	`abcd`
	`efgh`
		2-by-4 character array

`'it''s'`	`it's`	1-by-4 character array that includes an escaped `'` character. This array is written out as `'it''s'`.

Table 8-6: Input Syntax that Includes a Scaling Factor

Input	Array	Array Type

`1.0e–7 *`
`[`
`    0.1 0.2 ;`
`    0.3 0.4`
`]`
	`0.00000001  0.00000002`
	`0.00000003  0.00000004`
		2-by-2 square array. Note use of scaling factor in input.

Table 8-7: Illegal Input Syntax

Input	Array	Array Type
`[ 1 ; 2 3 ]`	Illegal. All rows must be same length.	Invalid array
`[ (1 + 2) 7; 4 5]`	Illegal. Using mathematical expressions in input files is not supported.	Invalid array

Using a Data File As Input

Let an input data file, `data`, contain the following array definition:

```
[ 1 2 3 ; 4 5 6 ]
```

Assume that a program called "`io`" reads an array from standard input and then writes it to standard output.

Passing the file `data` to the program `io`

```
io < data
```

produces this output:

```
[
    1 2 3 ;
    4 5 6
]
```

Using Stream I/O to Files

The preceding example demonstrates stream input from and output to the terminal. To read or write arrays from and to files, use the C++ class `ifstream` to create file input streams and `ofstream` to create file output streams.

For example, the code fragment shown below writes array `A` to a file and then reads the data from the file into array `B`.

Note that in order to run the code fragment, you need to insert the following at the top of the program:

```
#include <fstream.h>
```

The code fragment is as follows:

```
mwArray A = rand(5), B;
ofstream out_file("junk.txt", ios::out);
out_file << A << ends;
out_file.close();

ifstream in_file("junk.txt", ios::in);
in_file >> B;
```

A and B are now equal.

Using Streams for Interprocess Communication

You can use streams to facilitate sending an mwArray from one process to another. It is relatively simple to set up a socket-based mechanism that can send and receive strings between processes. Using the standard C++ strstream class, it is quite easy to write an mwArray into a string in one process, send the string to another process, and then read the mwArray from the string. Note that there is a form of the strstream constructor that binds a strstream to an already existing string. Use this form in the second process to read the mwArray.

Alternatively, you might use the shared memory routines on your system to share a string between two processes. Then, with a strstream in each process bound to the shared string, and a semaphore to control access to the shared memory, your two processes can send mwArray objects back and forth through the shared memory.

Last, and most ambitiously, you might define subclasses of istream and ostream to produce stream classes that manage the details of interprocess communication. With such classes defined, you could then send mwArray objects between processes simply by reading and writing from the streams.

Using File I/O Functions

The MATLAB C++ Math Library supports the following C and C++-style file I/O functions:

- `fprintf()`
- `fgetl()`
- `fgets()`
- `fopen()`
- `fclose()`
- `fscanf()`

The library's file I/O functions are similar to the ANSI standard C functions of the same name; they do, however, have several significant restrictions and extensions.

For example, the `fprintf()` function in the MATLAB C++ Math Library has two required input arguments and an unlimited number of optional input arguments. The first argument is the valid ID of an open file. The second is a format string that controls how the output data is formatted. In the library, both these arguments *must* be arrays.

The MATLAB C++ Math Library's version of `fprintf()` also processes the format string differently than the standard C++ `fprintf()`. Rather than requiring a format specifier for each input, it reuses the format string as necessary. The vectorized versions of `fprintf()` and `fscanf()` in the library take arrays as arguments, and repeatedly recycle their format strings through the arrays to produce the output or read the input. See the online *MATLAB C++ Math Library Reference* for complete details on each routine. "Accessing Online Reference Documentation" on page 1-7 describes how to access the Help Desk.

Specifying Library File I/O Functions

Because the MATLAB C++ Math Library file I/O functions have the same name as their C++ counterparts and because the types of their arguments are so similar, you must be careful to make sure you're calling the correct one.

This is particularly important with `fprintf()`. The type of the first argument to `fprintf()` is all important: if it is an array, the system calls the MATLAB

C++ Math Library function; if it is an integer, the system calls the standard C++ function. Consider this example:

```
mwArray file("foo.txt"), data=rand(4);
int fd = fopen(file);
fprintf(fd, "%f", data);
```

The system calls the standard C++ `fprintf()` function because the first argument passed to `fprintf()` is an integer. But this is almost certainly not what the author intended; the standard C++ `fprintf()` uses the format string to determine how many arguments it has. In this case, it will think there is a single argument and the program will crash because the standard `fprintf()` function does not understand `mwArray` objects.

The MATLAB C++ Math Library version of `sprintf()` requires that you pass an `mwArray` as its second argument. The other arguments may be passed as character strings.

Example – Using File I/O Functions (ex6.cpp)

The following example demonstrates how to use the `fopen()`, `fclose()`, `fprintf()`, `fgetl()`, and `fscanf()` routines. You can find the code for this example in the `<matlab>/extern/examples/cppmath` directory on UNIX systems and in the `<matlab>\extern\examples\cppmath` directory on PCs, where `<matlab>` represents the top-level directory of your installation. See "Building C++ Applications" on page 1-13for information about building and running the example program.

In the example, note the following:

- The `fopen()` routine returns the file ID as an array and accepts arrays for filename and mode arguments.
- The versions of `fprintf()` and `sprintf()` in the MATLAB C++ Math Library can take up to 32 arguments.
- `fgetl()` and `fgets()` work on ASCII files only.
- By default, `fopen()` opens files in read-only mode.

```
   // ex6.cpp

   #include <stdlib.h>
①  #include "matlab.hpp"

   int main(void)
   {
②      mwArray a("Alas, poor Yorick.  I knew him, Horatio.");
       mwArray b("Blow, wind, and crack your cheeks!");
       mwArray c("Cry havoc, and let slip the dogs of war!");
       mwArray d("Out, out, damned spot!");
       mwArray fid, r, a1, b1, c1, d1, mode("w"), sz, x, y;
       mwArray file("ex6.txt");

③  fid = fopen(file, mode);
④      fprintf(fid, "%s\n", a, b, c, d);
⑤      fclose(fid);

⑥      fid = fopen(file);
       a1 = fgetl(fid);
       b1 = fgetl(fid);
       c1 = fgetl(fid);
       d1 = fgetl(fid);
       cout << a1 << endl << b1 << endl << c1 << endl << d1 << endl;
       fclose(fid);

       fid = fopen(file, mode);
⑦      fprintf(fid, "%f ", magic(4), rand(4));
       fclose(fid);

       fid = fopen(file);
       sz = horzcat(4,4);
⑧      x = fscanf(fid, "%f ", sz);
       cout << x << endl;
       y = fscanf(fid, "%f ", sz);
       cout << y << endl;
       fclose(fid);

       return(EXIT_SUCCESS);
   }
```

The numbered items in the list below correspond to the numbered sections of code example:

1. Include header files. `matlab.hpp` declares the MATLAB C++ Math Library's data types and functions. `matlab.hpp` includes `iostream.h`, which declares the input and output streams `cin` and `cout`. `stdlib.h` contains the definition of `EXIT_SUCCESS`.

2. Declare local variables, including four string arrays. Notice that you can make a string array by passing a string to the `mwArray` constructor.

3. Open and create a file named `ex6.txt`. Unlike C++'s standard `fopen()`, the `fopen()` function in the MATLAB C++ Math Library takes one or two arguments, both `mwArray` objects. The first argument is the name of the file to open. The optional second argument is the mode in which to open the file: read (`"r"`) or write (`"w"`). Omitting the second argument implies read mode. `fopen()` returns an integer file id as an `mwArray` object.

   Note that you must pass arguments to `fopen()` as `mwArray` objects; otherwise, you will get the standard C version of `fopen()`.

4. Write the string arrays into the file. The `fprintf()` function in the C++ Math Library has two required input arguments and up to 30 optional input arguments. The first argument is the valid ID of an open file. The second is a format string that controls how the output data is formatted. The optional arguments follow the second argument.

   The MATLAB C++ Math Library's `fprintf()` processes the format string differently than the standard C++ `fprintf()`. Rather than requiring a format specifier for each input, it reuses the format string as necessary. Notice that the format string here is `"%s"`. Because there are fewer format specifiers than inputs, `fprintf()` applies this format to each input, `a`, `b`, `c`, and `d`.

5. Close the file. The standard C++ and MATLAB C++ Math Library `fclose()` functions behave similarly.

6. Reopen `ex6.txt`, and read in the four string arrays. `fgetl()` reads an entire line from a file, treating the line as a string. `fgetl()` reads from the current

position up to, but not including, the carriage return and, on the PC, the line feed that terminates the line.

A call to `fgetl()` skips over the end of line character(s) and never includes it in the string that it returns. Call `fgets()` if you need a string that contains the end-of-line character(s).

`fgetl()` and `fgets()` are designed to work on ASCII files only. Do not call them on binary files.

7 Print two numeric matrices, `magic(4)` and `rand(4)`, into the test file. Because the format string is `"%f"`, `fprintf()` prints each matrix as a row of floating-point numbers, applying the format string to each individual element of the matrices. `fprintf()` prints the matrix data in ASCII format.

8 Read the numeric matrices back from the test file. `fscanf()`'s first argument is the file id to read from; the second is a format string, and the third an optional size. As with `fprintf()`, the format string is recycled through the data as necessary. The third argument specifies the size and shape of the input data. In this case, the third argument is a 1-by-2 matrix containing the data `[4, 4]`. Given this size, `fscanf()` reshapes the input data into a 4-by-4 matrix.

Output

The program produces this output:

```
'Alas, poor Yorick.  I knew him, Horatio.'
'Blow, wind, and crack your cheeks!'
'Cry havoc, and let slip the dogs of war!'
'Out, out, damned spot!'
  [
        16         2         3        13 ;
         5        11        10         8 ;
         9         7         6        12 ;
         4        14        15         1
  ]


  [
    0.21896   0.93469   0.03457   0.00770 ;
```

```
  0.04704   0.38350   0.05346   0.38342 ;
  0.67887   0.51942   0.52970   0.06684 ;
  0.67930   0.83096   0.67115   0.41749
]
```

Importing and Exporting MAT-File Data

The MATLAB C++ Math Library provides two functions, `load()` and `save()`, that let you import `mwArray` variables from a MAT-file and export `mwArray` variables to a MAT-file. Because MATLAB also reads and writes MAT-files, you can use `load()` and `save()` to share data with MATLAB applications or with other applications developed with the MATLAB C++ or C Math Library.

A MAT-file is a binary, machine-dependent file. However, it can be transported between machines because of a machine signature in its file header. The MATLAB C++ Math Library checks the signature when it loads variables from a MAT-file and, if a signature indicates that a file is foreign (file was saved on a different architecture than the one on which it is being loaded), performs the necessary conversion.

Note The MATLAB C++ Math Library functions `save()` and `load()` implementations do not support all the variations of the MATLAB `load` and `save` syntax. In addition, the `load()` and `save()` implementations do not conform to the standard MATLAB C++ Math Library calling convention: they accept arguments that are not of type `mwArray` or `mwArray *`. The `load()` routine also allows output and input arguments to be interspersed.

Exporting Array Data to a MAT-File

Using `save()`, you can save the data within `mwArray` variables to disk. The prototype for `save()` is:

```
void save(const mwArray &file, const char* mode,
     const char* name1, const mwArray &var1,
     const char* name2=NULL, const mwArray &var2=mwArray::DIN,
     .
     .
     .
     const char* name16=NULL, const mwArray &var16=mwArray::DIN );
```

`file` contains the name of the MAT-file; `mode` points to a string that indicates whether you want to overwrite or update the data in the file. You must pass at least one pair of arguments indicating the name you want to assign to the data you're saving and the address of the `mwArray` variable that you want to save:

- You must name each mwArray variable that you save to disk. A name can contain up to 32 characters.
- You can save up to 16 variables in a single call to save().
- There is no call that globally saves all the variables in your program or in a particular function.
- The name of a MAT-file must end with the extension .mat. The library appends the extension .mat to the filename if you do not specify it.
- You can either overwrite or append to existing data in a file. Pass "w" to overwrite, "u" to update (append), "w4" to overwrite using V4 format. A second version of the save() function allows you to omit the mode argument; the default is to overwrite the data.
- The file created is a binary MAT-file, not an ASCII file.

Importing Array Data from a MAT-File

Using load(), you can read in mwArray data from a binary MAT-file. The prototype for load() is:

```
void load( const mwArray &file,
    const char* name1, mwArray *var1,
    const char* name2=NULL, mwArray *var2=NULL,
    .
    .
    .
    const char* name16=NULL, mwArray *var16=NULL );
```

file contains the name of the MAT-file. You must pass at least one pair of arguments indicating the name of a variable that you want to load and a pointer to an mwArray variable that will receive the data:

- You must indicate the name of each mwArray object that you want to load.
- You can load up to 16 mwArray objects in one call to load().
- There is no call that globally loads all variables from a MAT-file.
- You do not have to allocate space for the incoming mwArray. load() allocates the space required based on the size of the variable being read.
- You must specify a full path for the file that contains the data. If you do not specify the .mat extension, the library automatically appends it to the filename.

- You must load data from a binary MAT-file, not an ASCII MAT-file.

Note Be sure to transmit MAT-files in binary file mode when you exchange data between machines.

For more information on MAT-files, consult the *MATLAB Application Program Interface Guide*.

Example – Using load() and save() (ex7.cpp)

This example demonstrates how to use the functions `load()` and `save()` to write your data to a disk file and read it back again. You can find the code for this example in the `<matlab>/extern/examples/cppmath` directory on UNIX systems and in the `<matlab>\extern\examples\cppmath` directory on PCs, where `<matlab>` represents the top-level directory of your installation. See "Building C++ Applications" on page 1-13for information about building and running the example program.

In the example, note the following:

- You must name the variables when you save them to a MAT-file.
- You must specify the name of the variable you want to read from a MAT-file.
- `load()` and `save()` do not conform to the standard MATLAB C++ Math Library calling convention:
  - Not all arguments are of type `mwArray` or `mwArray *`.
  - Output and input arguments to `load()` are interspersed.
- MAT-files must have the three-letter extension `mat`. If you do not specify the `.mat` extension, `load()` and `save()` automatically add it.

```
    // ex7.cpp

    #include <stdlib.h>
1   #include "matlab.hpp"

    int main(void)
    {
        try {
2           mwArray x, y, z, a, b, c;

3           x = rand(4,4);
            y = magic(7);
            z = eig(x);

            // Save (and name) the variables.
4           save("ex5.mat", "x", x, "y", y, "z", z);

            // Load the named variables.
5           load("ex5.mat", "x", &a, "y", &b, "z", &c);

            // Check to be sure the variables are equal.
6         if (tobool(a == x) && tobool(b == y) && tobool(c == z))
            {
                cout << "Success: all variables equal." << endl;
            }
            else
            {
                cout << "Failure: loaded values not equal to
                         saved values." << endl;
            }
        }
        catch (mwException &ex) {
            cout << ex << endl;
        }
       return(EXIT_SUCCESS);
    }
```

The numbered items in the list below correspond to the numbered sections of code example:

1 Include header files. `matlab.hpp` declares the MATLAB C++ Math Library's data types and functions. `matlab.hpp` includes `iostream.h`, which declares the input and output streams `cin` and `cout`. `stdlib.h` contains the definition of `EXIT_SUCCESS`.

2 Declare and initialize variables. `x`, `y`, and `z` are written to the MAT-file using `save()`. `a`, `b`, and `c` store the data read back from the MAT-file by `load()`.

3 Assign data to the variables that will be saved to a file. `x` stores a 4-by-4 array that contains randomly-generated numbers. `y` stores a 7-by-7 magic square. `z` contains the eigenvalues of `x`.

4 Save three variables to the file `ex5.mat`. In one call to `save()`, you can save up to 16 variables to the file identified by the first argument. Subsequent arguments come in pairs: the first argument in the pair (a string) labels the variable in the file; the contents of the second argument, an `mwArray`, is written to the file.

An additional signature for `save()` allows you to specify a mode for writing to the file: `"w"` for overwrite, `"u"` for update (append), and `"w4"` for overwrite in version 4 format. Without the mode argument, as in this example, `save()` overwrites the data.

Note that you must provide a name for each variable you save. When you retrieve data from a file, you must provide the name of the variable you want to load. You can choose any name for the variable; it does not have to correspond to the name of the variable within the program.

5 Load the named variables from the file `"ex5.mat"`. Note that the function `load()` does not follow the standard C++ Math Library calling convention where output arguments precede input arguments. The output arguments, `a`, `b`, and `c`, are interspersed with the input arguments.

Pass arguments in this order: the filename and then the name/variable pairs themselves. You can read in up to 16 `mwArray` objects at a time. An important difference between the syntax of `load()` and `save()` is the type of the variable portion of each pair. Because you're loading data into a variable, `load()` needs the address of the variable: `&a`, `&b`, `&c`. `a`, `b`, and `c` are output arguments whereas `x`, `y`, and `z` in the `save()` call are input arguments.

Notice how the name of the output argument does not have to match the name for the variable stored in the file.

6 Compare the data loaded from the file to the original data that was written to the file. `a`, `b`, and `c` contain the loaded data; `x`, `y`, and `z` contain the original data. The calls to `tobool()` are necessary because C++ requires that the conditional expression of an `if` statement be a scalar Boolean. `tobool()` reduces the rank of its argument to a scalar, and then returns a Boolean value.

Output

When run, the program produces this output:

```
Success: all variables equal.
```

9

Translating from MATLAB to C++

Differences Between C++ and MATLAB

Most MATLAB expressions translate into C++ with no effort — very often the MATLAB and C++ are identical. There are some differences in syntax, of course, but it is important to realize that the C++ interface is substantially the same as the M-file interface.

MATLAB and C++ syntax are identical in the following four areas:

- Simple function calls that have one output and one or more inputs
- Arithmetic expressions consisting entirely of matrix operations (+, *, /, –)
- Array indexing expressions that don't use the colon operator, cell array indexing, or structure indexing
- Assignment statements, including assignment with a standard indexing expression on the left-hand side

The differences between C++ and MATLAB are discussed in detail below. More space is devoted to differences than similarities, not because there are more differences, but because the differences are more likely to cause confusion.

Programming in C++ differs from programming in MATLAB in five important areas:

- "Syntax"
- "Variable Declaration"
- "Function Calling Conventions"
- "Control Structure"
- "Logical Values"
- "Name Conflicts with Standard C Library Functions"
- "Example Program: Rewriting roots.m in C++ (ex8.cpp)"

Syntax

The syntax of C++ places several restrictions on the MATLAB C++ Math Library. In general, these restrictions do not mean that functionality is missing from the library, but rather that you access the functionality differently than you would in MATLAB.

C++ restricts the syntax of the library in these ways:

- You cannot construct arrays with MATLAB's [ ] array construction syntax. Instead, you call a constructor of the `mwArray` class, an array creation function, or the `vertcat()` and `horzcat()` functions.
- The `:` (colon) operator is unavailable in all of its forms. The functional equivalents `colon()` and `ramp()` replace it.
- The mathematical operators `.*`, `.\`, `./`, `.^` and `\` are not valid C++ operators. In the MATLAB C++ Math Library, function calls access the same functionality.
- The `'` (quote) and `.'` (dot quote) operators are unavailable. The `transpose()` and `ctranspose()` functions replace them.
- The `{}` cell array indexing operator is unavailable. The `mwArray::cell()` performs indexing into cell arrays.
- The `.` operator for structure indexing is unavailable. The `mwArray::field()` performs indexing into structures.

Each of these differences is explained in more detail later in this chapter.

Variable Declaration

In addition to requiring different syntax, C++ insists that you declare all variables explicitly before using them. The declarations do not have to appear at the top of a function as they do in C, but may be interspersed throughout your code.

Declare array variables as type `mwArray`. Note that in general `mwArray` objects have value semantics in the MATLAB C++ Math Library. They are passed by value to functions; they are not modified by functions; they are returned by value.

To modify the value of an `mwArray` object within a function, pass the `mwArray` object to that function by reference, either as a pointer (`mwArray *`) or a reference (`mwArray &`).

Note If you are a user of the MATLAB Application Program Interface, don't confuse the type `mwArray` with the `mxArray` type used in the Application Program Interface Library. Do not declare array variables as type `mxArray*` unless you really want a pointer to an `mxArray`.

Function Calling Conventions

MATLAB and C++ have different function calling conventions. In MATLAB, a function declaration establishes a function's name. The declaration says nothing about the number and type of the inputs and outputs to the function. In C++, a function declaration does specify the number and type of the input arguments and the type of the return value.

In addition, in C++ a function can return at most one value whereas MATLAB functions can return more than one value. Functions in the MATLAB C++ Math Library emulate their MATLAB counterparts that have multiple return values by returning one value as the return from the function and storing the rest of the values in output arguments supplied by the caller.

For complete details on the library's calling conventions, see "How to Call C++ Library Functions" in Chapter 5.

Control Structure

Both C++ and MATLAB support `if`-statements, `for`-loops, and `while`-loops. The primary difference between the C++ and MATLAB versions of these constructs is syntactical. For instance, in MATLAB the `end` keyword terminates a `for`-loop; in C++ braces surround the body of the loop.

There are two subtle functional differences, however, between the C++ and MATLAB `for`-loop constructs, both concerning the index for the loop. In C++ you can modify the `for`-loop index and the bounds for the index in the middle of the loop. In MATLAB the interpreter ignores any modifications to the loop index or its bounds.

The second subtle difference between the two `for`-loops is the final value of the index variable. When a MATLAB loop terminates, the index variable is equal to the loop's upper bound. When a C++ loop terminates, the index variable is typically one greater than the loop's upper bound. However, this is not true of C++ code generated by the MATLAB Compiler.

Refer to your C++ reference manual for more information on how `for`-loops work in C++.

Logical Values

In MATLAB, a logical value is either a logical scalar or an array of logical values. You create a logical array by calling the `logical()` function or by using a relational operator to compare two arrays. In C++, logical values are always scalars. A 1-by-1 `mwArray` object can be cast to a scalar. When an array object appears where a logical value is expected, C++ automatically attempts to cast the array to a scalar. This casting operation fails (raises an exception) if the array is not 1-by-1.

When a relational operation between arrays appears where a scalar Boolean value is required, you must use the MATLAB C++ Math Library function `tobool()` to reduce the result of the operation to a scalar Boolean. `tobool()` reduces any real or complex array to a Boolean true or false result. If you pass `tobool()` an empty array, it returns false.

For example, to test if every element in an array `A` is nonzero, write:

```
if (tobool(A != 0))
{
    // test succeeded, do something
}
```

`if` and `while` statements in C++ require you to use these functions. Because the relational operators (<, >, <=, >=, == and `!=`) each return an array of logical values in both MATLAB and C++, it is necessary, when using the result of one of these operators in an `if` or `while` statement, to wrap it with a call to `tobool()`.

There is one exception to this rule: if an array is a scalar, `tobool()` is unnecessary, since the compiler will attempt to convert the array, by default, to a double. However, if the array is not a scalar, this conversion fails at runtime and throws an exception.

Name Conflicts with Standard C Library Functions

Some functions in the standard C math library, `libm`, that is supplied with every C and C++ compiler have the same names as functions in the MATLAB C++ Math Library. The exact number of functions in conflict varies by platform. The MATLAB C++ Math Library uses two methods to resolve these name conflicts: argument casting and function renaming.

The MATLAB C++ Math Library renames some functions so that the library function is unique. For other functions, you must cast the argument passed to the function to the type expected by the MATLAB function.

Casting an Argument to Avoid a Name Conflict

The most common naming conflicts between the two libraries occur with the trigonometric functions (`sin()`, `cos()`, `tan()`, etc.), the logarithmic and exponential functions (`log()`, `log10()` and `exp()`), and several miscellaneous functions like `sqrt()` and `abs()`. These duplicate functions cause a problem when invoked with either a C++ `int` or `double` scalar argument. They do not cause a problem when they're invoked with an `mwArray` argument.

For example, when the C++ compiler sees a call such as `sqrt(–1)`, it generates a call to the `sqrt()` defined in the standard C math library rather than the `sqrt()` defined by the MATLAB C++ Math Library. The C runtime library conforms to the IEEE standard: the square root of a negative number is `NaN`. However, the range of the MATLAB C++ Math Library's `sqrt()` routine extends into the complex plane, so that it returns the complex number `i` when called with `–1`.

Because C++ does not allow a function name to be overloaded on the basis of return type alone, it is not possible to add functions to the MATLAB C++ Math Library that take scalars and return `mwArray`'s and thus distinguish between a function in the standard C math library and one in the MATLAB C++ Math Library. Renaming all the MATLAB functions like `sqrt()` and `abs()` would only cause confusion. Therefore, to avoid this problem, we recommend that you never invoke these functions with a scalar argument.

For example, if you need to determine the square root of a negative quantity, first create an array and assign the negative number to it. Then call `sqrt()` on the array:

```
mwArray a = –5;
```

```
sqrt(a);
```

You can also use a cast:

```
sqrt((mwArray)-5);
```

or an explicit constructor call:

```
sqrt(mwArray(-5));
```

The last two techniques are the most succinct.

Renaming Functions to Avoid a Name Conflict

Casting arguments cannot resolve all the naming conflicts between the two libraries. For example, the MATLAB C++ Math Library functions `char` and `double` conflict with C++ data types. The library's `clock()` function doesn't take any arguments and thus can't be overloaded. Whenever a MATLAB function name conflicts with a C++ keyword, type, or built-in function, the MATLAB C++ Math Library appends `_func` to its name.

This table lists the functions in the library that have been renamed.

Table 9-1: Renamed Functions in the MATLAB C++ Math Library

MATLAB Name	C++ Math Library Name
`and`	`and_func`
`bitand`	`bitand_func`
`bitor`	`bitor_func`
`char`	`char_func`
`clock`	`clock_func`
`double`	`double_func`
`not`	`not_func`
`or`	`or_func`
`pascal`	`pascal_func` (PC only)
`quad`	`quad_func`

Table 9-1: Renamed Functions in the MATLAB C++ Math Library (Continued)

MATLAB Name	C++ Math Library Name
std	std_func
struct	struct_func
union	union_func
xor	xor_func

Example Program: Rewriting roots.m in C++ (ex8.cpp)

The `roots()` function finds the roots of a polynomial. The M-file `roots.m` contains the source of the `roots()` function. This example shows how to translate `roots.m` into C++. The translation keeps the C++ function as similar as possible to the M-function, primarily to demonstrate how easy it is to write MATLAB-like code in C++. This means that the C++ code is not as efficient as it could be, but the example does show that the C++ code is as simple to write as a MATLAB M-file.

The M-File roots() Function

The C++ `example_roots()` function is a translation of the M-file `roots()` function. For purposes of comparison, `roots.m` is reproduced below. Not counting the comments or the main routine, the C++ code is only four lines longer than the M-code. Two of the extra lines are used for declaring variables and the other two for including header files.

MATLAB M-file code for `roots()`:

```
function r = roots(c)
%ROOTSFind polynomial roots.
%   ROOTS(C) computes the roots of the polynomial whose
%   coefficients are the elements of the vector C. If C has N+1
%   components, the polynomial is C(1)*X^N + ... + C(N)*X +
%   C(N+1).
%
%   See also POLY.
%   J.N. Little 3-17-86
%   Copyright (c) 1984-97 by The MathWorks, Inc.
% ROOTS finds the eigenvalues of the associated companion matrix.

n = size(c);
if ~sum(n <= 1)
    error('Must be a vector.')
end
n = max(n);
c = c(:).';    % Make sure it's a row vector

% Strip leading zeros and throw away.  Strip trailing zeros,
% but remember them as roots at zero.
inz = find(abs(c));
nnz = max(size(inz));
if nnz ~= 0
    c = c(inz(1):inz(nnz));
    r = zeros(n-inz(nnz),1);
else
    r = [];
end

% Polynomial roots via a companion matrix
n = max(size(c));
a = diag(ones(1,n-2),-1);
if n > 1
    a(1,:) = -c(2:n) ./ c(1);
end
r = [r;eig(a)];
```

The C++ roots() Function

The example is divided into two parts. The first part shows the main program, which sets up the problem and invokes the example version of `roots()`: the `example_roots()` function. The second part contains the `example_roots()` function. In the C++ source file, the order of the parts is reversed. The parts are reordered here for clarity.

You can find the code for this example in the `<matlab>/extern/examples/cppmath` directory on UNIX systems and in the `<matlab>\extern\examples\cppmath` directory on PCs, where `<matlab>` represents the top-level directory of your installation. See "Building C++ Applications" on page 1-13 for information about building and running the example program.

In the example, note the following:

- Programs written using the MATLAB C++ Math Library look very much like MATLAB M-files. The syntax of the two is very similar.
- The MATLAB C++ Math Library supports most of MATLAB's operators. Those that are not supported as operators can be accessed via function calls.
- The functions `colon()` and `ramp()` in the MATLAB C++ Math Library replace the MATLAB colon operator.
- A function that modifies its input arguments, or uses them as a temporary variables, must not declare those arguments `const`.
- The default `mwIndex` constructor produces an `mwIndex` object that acts, when used as a subscript, like a colon in MATLAB.
- The C++ `if`-statement, unlike the MATLAB `if` statement, requires that any matrices tested be reduced in rank to scalars by calls to `any()` or `all()`. See the online *MATLAB C++ Math Library Reference* for further explanation of the functions. "Accessing Online Reference Documentation" on page 1-7 describes how to access the Help Desk.
- Calling the `error()` function throws an `mwRuntimeException` exception. `mwRuntimeException` is a subclass of `mwException`.
- `vertcat()` vertically concatenates two or more matrices. `horzcat()` behaves like `vertcat()` except performs horizontal concatenation.
- Use the C++ operator `!` to invert the truth value of a logical scalar `mwArray`; use the MATLAB C++ Math Library operator ~ to invert the truth value of a logical array. Do not apply `!` to arrays.

```
    // Call example_roots() and the library's roots().

    int main(void)
    {
①       // Static array of doubles used to initialize the matrices.
        static double input[] = { 1, -6, -72, -27 };

②       // Declare three matrices, one with initial values.
        mwArray x(1, 4, input), result, verify;

③       // Call our version of roots().
        result = example_roots(x);

④       // Call the MATLAB C++ Math Library roots.
        verify = roots(x);

        // Print the input and output matrices from example_roots().
⑤       cout << "x = " << endl << x << endl;
        cout << "example_roots(x) = " << endl << result << endl;

⑥
        // Check to see if the answer is equal to the real roots().
        if (tobool(result == verify))
            cout << "Success!" << endl;

        return(EXIT_SUCCESS);
    }
```

The numbered items in the list below correspond to the numbered sections of code example:

The main program is straightforward, so the explanations below are brief. If you have difficulty understanding this section of the example, refer to "Example Program: Writing Simple Functions (ex4.cpp)" in Chapter 2.

1 Declare the static variable used for array initialization. The elements of the C++ array are specified in the column-major order required by the library.

2 Declare and initialize three matrices. `x`, a row vector (one row, four columns), is the input matrix. `result` and `verify` are initially null matrices.

3 Call the `example_roots()` function and place the return value in the matrix `result`.

4 Call the MATLAB C++ Math Library's version of `roots()` and store the return value in the matrix `verify`. `verify` will be used to confirm that the rewriting of `roots` produces the correct result.

5 Print the input to `example_roots()`. Print the output from `example_roots()`.

6 Verify the result. The matrix `verify` contains the correct result. The `mwArray` class provides an overloaded `operator==()`, which makes comparing two matrices for equality easy.

The second part of the example is the `example_roots()` function itself. This function is part of the same file as the main program shown above.

```
    #include <stdlib.h>
(1) #include "matlab.hpp"

    // EXAMPLE_ROOTS(C) computes the roots of the polynomial whose
    // coefficients are the elements of the vector C. If C has N+1
    // components, the polynomial is C(1)*X^N + ... + C(N)*X + C(N+1).
(2) mwArray example_roots(mwArray c)
    {
(3)     mwArray n, inz, nnz, r, a;
        mwIndex icolon;

        // Make sure number of dimensions is not greater than 1.
(4)     n = size(c);
        if (all(n > 1.0))
            error("Must be a vector");

(5)     n = max(n); // <5>
        c = transpose(c(icolon)); // Make sure it's a row vector.
(6)     inz = find(abs(c));       // Find all nonzero elements.
        nnz = max(size(inz));     // Count nonzero elements.

        // Test all elements against zero.
(7)     if (!(nnz == 0.0))
        {
          c = c(ramp(inz(1), inz(nnz))); //Strip leading/trailing 0's
            r = zeros(n - inz(nnz), 1);    //Remember trailing 0's
        }

        //  Polynomial roots via a companion matrix
(8)     n = max(size(c));  // Size of the largest dimension of c
       a = diag(ones(1, n - 2.0), -1.0); // Create a row vector of 1's.
(9)     if (n > 1.0)
            a(1,icolon) = -c(ramp(2, n)) / c(1);
(10)    r = vertcat(r, eig(a));

        return r;
    }
```

The numbered items in the list below correspond to the numbered sections of code example:

1 Include header files. `matlab.hpp` declares the MATLAB C++ Math Library's data types and functions. `matlab.hpp` includes `iostream.h`, which declares the input and output streams `cin` and `cout`. `stdlib.h` contains the definition of `EXIT_SUCCESS`.

2 Declare the C++ function `example_roots()`. We can't use the name `roots()` because the MATLAB C++ Math Library defines a `roots()` function with exactly the same number and type of input and output arguments. `example_roots()` has one input and one output, both of which are matrices. The input argument is not declared `const` because `example_roots()` stores temporary results in `c`.

3 Declare the matrix and index variables. All variables used in the program must be declared before being used. Since `icolon` is declared using the default `mwIndex` constructor, this variable acts like MATLAB's `:` operator when used in array indexing expressions. Declaring a variable like this is more efficient than repeatedly calling the `colon()` function. See "Programming Efficient Indices" in Chapter 4 for more information.

4 Check for valid inputs. Determine the size of the input matrix and store the result (a 1-by-2 matrix, i.e., a vector) in the variable `n`. At least one of the dimensions of the input matrix must be equal to one, that is, the matrix must be a vector. Report an error to the user and terminate if the matrix is malformed. Calling the `error()` function causes a runtime exception to be thrown.

5 Determine the size of the largest dimension of the input matrix. `n` is now a 1-by-1 matrix: a scalar. Use the colon operator (here represented by the variable `icolon`) to extract the input matrix into a column vector. Transpose this vector to get a row vector. The root-finding algorithm below requires that the matrix be a row vector. You could improve the efficiency here by testing the dimensions of the input matrix and transforming them only when necessary.

6 Find all the nonzero elements of the input matrix. Store the result in a vector. Then count the number of nonzero elements. Since `inz` is a vector, `size()` returns a vector `[ 1 N ]`, where `N` is the length of the vector. `N` is the count of elements in `inz`.

7 Strip leading zeros and delete them. Strip trailing zeros, but remember them as roots at zero. It is possible that the input matrix was full of zeros. In this case, `find()` will have returned a null matrix and `nnz` will be equal to 0. Note that wrapping the logical expression with `all()` is not necessary in this case, since `nnz` is known to be a scalar.

If the input matrix did not contain all zeros, `inz` contains the nonzero elements. Replace the input matrix with a vector `1,2,...,N` where `N` is the number of nonzero elements originally in the input matrix. The result from the call to `ramp()` goes from the first nonzero index to the last.

Set `r` to a column vector of zeros, with one row for each trailing zero element of the input matrix. The arguments to the `zeros()` function are row count and column count.

8 Determine the size of the largest dimension of the input matrix, which may have changed, since the zero elements have been removed from the matrix. Use this size to form a diagonal matrix, with 1's on the -1 diagonal.

9 If `c` is not empty, replace the first row of matrix `a` with the vector resulting from dividing the negative of the `2,...,N` elements of `c` (enumerated by the call to `ramp()`) by `c(1)`. This type of assignment, where an indexing expression appears on the left-hand side, is the only way to modify the contents of a matrix.

10 Vertically concatenate the matrices `r` and `eig(a)`. The number of columns in both matrices must be the same. The rows of `eig(a)` are placed below the rows (in this case, the single row) of `r`. Reassign the result to `r`. Return the matrix `r`. Unlike MATLAB, C++ requires an explicit return statement.

Output

The program produces this output:

```
x =
  [
        1       -6      -72      -27
  ]


example_roots(x) =
```

```
  1.0e+01 *

  [
    1.21229 ;
   -0.57345 ;
   -0.03884
  ]


Success!
```

mwArray Class Interface

Introduction

The mwArray class public interface (those functions you can call directly) is relatively small, consisting of constructors and a destructor, overloaded new and delete operators, one user-defined conversion, indexing operators and functions, the assignment operator, input and output operators, and array size query routines. Since the mwArray public interface is relatively small, it is not likely to require extensive modification in future versions of the library.

The mwArray's public interface does not contain any mathematical operators or functions. This does not mean, of course, that these operators and functions are not available. To the contrary, the MATLAB C++ Math Library contains more than 400 mathematical routines. These routines use the mwArray class interface; however, they are not member functions.

Both the users of a library and its developers benefit from a relatively small, static interface for the mwArray class. The smaller interface is easier to understand than a larger one simply because it contains fewer routines. Similarly, the uniform interface for the mathematical functions, in which the rules are the same for all functions, is easier to learn. By virtue of being excluded from the interface of the mwArray class, the mathematical routines gain a uniformity of interface.

For example, consider the functions transpose() and eig(). An argument could be made that transpose() should be a member function of mwArray, for then it would be invoked by the syntax A.transpose(), which is quite natural to both mathematicians and C++ programmers. However, the case for eig() as a member function is much weaker. eig() can be called with several different types of arguments. In at least one of the combinations, [V, D] = eig(A, B), it is not clear which, if any, of the arguments is the "object" on which eig() is invoked. Furthermore, because of the way in which multiple return arguments are implemented in the MATLAB C++ Math Library, picking an arbitrary input argument to act as the "object" produces a confusing interleaving of input and output arguments.

This problem arises with many functions in the MATLAB C++ Math Library, making them inappropriate mwArray member functions. Rather than divide the mathematical routines into two groups – member functions and nonmember functions – we decided that a uniform interface to the mathematical functions was more important than dogmatically adhering to the object.function()

syntax of C++. Therefore, none of the MATLAB mathematical routines are member functions of `mwArray`.

Constructors

The mwArray interface provides many useful constructors. You can construct an mwArray object from the following types of data: a numerical scalar, an array of scalars, a string, an mxArray *, or another mwArray object. This table lists the most commonly used constructors.

Table 10-1: mwArray Constructors

Constructor	Creates	Example
`mwArray()`	Uninitialized array	`mwArray A;`
`mwArray(const char *)`	String array	`mwArray A("MATLAB Rules");`

`mwArray(int32, int32, double*, double*)`	Complex array	`double real[] = { 1, 2, 3, 4 };`
		`double imag[] = { 5, 6, 7, 8 };`
		`mwArray A(2,2,real,imag);`

`mwArray(const mwArray&)`	Copy of input array	`mwArray A = rand(4);` `mwArray B(A);`
`mwArray(const mxArray *)`	Copy of mxArray*	`mxArray *m = mlfScalar(1);` `mwArray mat(m);`
`mwArray(double, double, double)`	Ramp	`mwArray A(1.2, 0.1, 3.5);`
`mwArray(int32, int32, int32)`	Integer ramp	`mwArray A(1, 2, 9);`
`mwArray(const mwSubArray&)`	Array from subarray (used in indexing)	`mwArray A = rand(4);` `mwArray B(A(3,3));`
`mwArray(double)`	Scalar double array	`mwArray A(17.5);`
`mwArray(int)`	Scalar integer array	`mwArray A(51);`

Each constructor is described below:

- `mwArray()`

  Create an uninitialized array. An uninitialized array produces warnings when passed to MATLAB C++ Math Library functions. If an array is created using this default constructor, a value must be assigned to it before passing it to a MATLAB C++ Math Library function.

  To create an empty double matrix that corresponds to `[]` in MATLAB, use the function `empty()`.
- `mwArray(const char *str)`

  Create an array from a string. The constructor copies the string.
- `mwArray(int32 rows, int32 cols, double *real, double *imag = 0)`: Create an `mwArray` from either one or two arrays of double-precision floating-point numbers. If two arrays are specified, the constructor creates a complex array; both input arrays must be the same size. The data in the input arrays must be in column-major order, the reverse of C++'s usual row-major order. See Chapter 3, "Working with MATLAB Arrays", for more information on the difference between row- and column-major data order. This constructor copies the input arrays.

  Note that the last argument, `imag`, is assigned a value of zero in the constructor. `imag` is an optional argument. When you call this constructor, you do not need to specify the optional argument. Refer to a C++ reference guide for a more complete explanation of default arguments.
- `mwArray(const mwArray &mtrx)`

  Copy an `mwArray`. This constructor is the familiar C++ copy constructor, which copies the input array. For efficiency, this routine does not actually copy the data until the data is modified. The data is referenced through a pointer until a modification occurs.
- `mwArray(const mxArray *mtrx)`

  Make an `mwArray` from an `mxArray *`, such as might be returned by any of the routines in the MATLAB C Math Library or the Application Program Interface Library. This routine does *not* copy its input array, yet the destructor frees it; therefore the input array must be allocated on the heap. In most cases, for example, with matrices returned from the Application Program Interface Library, this is the desired behavior.

- `mwArray(double start, double step, double stop)`

  Create a ramp. This constructor operates just like the MATLAB colon operator. For example, the call `mwArray(1, 0.5, 3)` creates the vector `[ 1, 1.5, 2, 2.5, 3 ]`.

- `mwArray(int32 start, int32 step, int32 stop)`

  Create an integer ramp.

- `mwArray(const mwSubArray & a)`

  Create an `mwArray` from an `mwSubArray`. When an indexing operation is applied to an array, the result is not another array, but an `mwSubArray` object. An `mwSubArray` object remembers the indexing operation. Evaluation of the operation is deferred until the result is assigned or used in another expression. This constructor evaluates the indexing operation encoded by the `mwSubArray` object and creates the appropriate array.

- `mwArray(double)`

  Create a 1-by-1 `mwArray` from a double-precision floating-point number.

- `mwArray(int)`

  Create an `mwArray` from an integer.

See Chapter 3, “Working with MATLAB Arrays” for more examples of how to use constructors.

Indexing and Subscripts

The `mwArray` interface supports multidimensional indexing, including cell array and structure indexing:

- The operator `()` supports multidimensional indexing into arrays, including access to cells or structures that make up an array.
- The member function `cell()` supports indexing into the contents of a cell.
- The member function `field()` supports indexing into the contents of a structure field.

See Chapter 4, "Indexing into Arrays" for examples of how these routines are used.

Array Indexing

Array indexing is implemented through the interaction of three classes: `mwArray`, `mwSubArray`, and `mwIndex`. When applied to an `mwArray`, `operator()` returns an `mwSubArray`. The `mwSubArray` "remembers" the indexing operation and defers evaluation until the result is either assigned or referred to.

You can pass an integer, double, `mwArray`, the contents of a cell or structure field, or an indexing expression as an argument to `operator()`.

The `mwArray` class interface contains a series of `operator()` member functions that support *n*-dimensional indexing. Several of the functions are listed here.

This pair of `operator()` member functions supports one-dimensional indexing and indexing into arrays with more than 32 dimensions. The second non-`const` signature supports calls that are targets of the assignment operator and modify the contents of an array.

```
mwArray operator()(const mwVarargin &a) const;
mwSubArray operator()(const mwVarargin &a);
```

This pair of `operator()` member functions supports two-dimensional indexing The second non-`const` signature supports calls that are valid targets for the assignment operator.

```
mwArray operator()(const mwArray &a1, const mwArray &a2) const;
mwSubArray operator()(const mwArray &a1,
                      const mwArray &a2);
```

This pair of operator() member functions supports the maximum number of arguments. To index into more than 32 dimensions, you must construct an mwVarargin object.

```
mwArray operator()(const mwArray &a1,
                   const mwArray &a2,
                   const mwArray &a3,
                  .
                  .
                  .
                   const mwArray &a32) const;
```

```
mwSubArray operator()(const mwArray &a1,
                      const mwArray &a2,
                      const mwArray &a3,
                     .
                     .
                     .
                      const mwArray &a32);
```

Cell Content Indexing

These two versions of the cell() member function let you index into the contents of a cell. For example, A.cell(1,2) refers to the contents of the cell in the second column of the first row in an array A.

The cell() member functions follow the library convention for varargin functions. You can pass up to 32 arguments to the functions. To index into more than 32 dimensions, you must construct an mwVarargin object and pass it as the first argument. That object allows you to reference an additional 32 arguments, the first of which can again be an mwVarargin object.

The second non-const signature supports calls that are targets of the assignment operator and modify the contents of a cell.

```
mwArray cell(const mwVarargin &RI1,
             const mwArray &OI2=mwArray::DIN,
             const mwArray &OI3=mwArray::DIN,
             .
             .
             .
             const mwArray &OI32=mwArray::DIN ) const;
```

```
mwSubArray cell(const mwVarargin &RI1,
             const mwArray &OI2=mwArray::DIN,
             const mwArray &OI3=mwArray::DIN,
             .
             .
             .
             const mwArray &OI32=mwArray::DIN );
```

Structure Field Indexing

The two versions of the `field()` member function let you reference the field of a structure. For example, `A.field("name")` accesses the contents of the field called `name` within the structure `A`.

The second non-`const` signature supports calls that are targets of the assignment operator and modify the contents of a field.

```
mwArray field(const char *fieldname) const;
mwSubArray field(const char *fieldname);
```

User-Defined Conversions

There is only one user-defined conversion: from an mwArray to a double-precision floating-point number. This conversion function only works if the mwArray is scalar (1-by-1) and noncomplex:

```
operator double() const;
```

Memory Management

Overloading the operators `new` and `delete` provides the necessary hooks for user-defined memory management. The MATLAB C++ Math Library has its own memory management scheme (See “Memory Management” in Chapter 7 for details).

If this scheme is inappropriate for your application, you can modify it. However, you should not do so by overloading `new` and `delete`, because the `mwArray` class already contains overloaded versions of these operators:

- `void *operator new(size_t size)`
- `void operator delete(void *ptr, size_t size)`

Operators

In addition to the indexing operators, there are three additional operators in the mwArray interface. The first two operators, << and >>, are used for stream input and output. Technically, these stream operators are not member functions; they are friend functions:

- `friend inline ostream& operator<<(ostream &os, const mwArray&)`

  Calling this operator inserts an mwArray object into the given stream. If the stream is cout, the contents of the mwArray object appear on the terminal screen or elsewhere if standard output has been redirected on the command line. This function simply invokes Write() as described below.

- `friend inline istream& operator>>(istream &is, mwArray&)`

  This is the stream extraction operator, capable of extracting, or reading, an mwArray from a stream. The stream can be any C++ stream object, for example, standard input, a file, or a string. This function simply invokes Read() as described below. "Using Array Stream I/O" in Chapter 8 describes the syntax of the input format.

Note that the >> and << operator functions do not read and write MAT-files.

The stream operators call Read() and Write(), mwArray public member functions.

Note Write() writes arrays in exactly the format that Read() reads them. An array written by Write() can be read by Read(). These functions read and write full double arrays only. Read() does not read sparse arrays, cell arrays, or structures. Use MAT-files to save and restore these arrays.

- `void Read(istream&)`

  Reads an mwArray from an input stream. An array definition consists of an optional scale factor and asterisk, *, followed by a bracket [, one or more semicolon-separated rows of double-precision floating-point numbers, and a closing bracket]. "Using Array Stream I/O" in Chapter 8 describes the input format in more detail.

- `void Write(ostream&, int32 precision =5, int32 line_width =75) const`

  Formats `mwArray` objects using the given precision (number of digits) and line width, and then writes the objects into the given stream. `operator<<()` uses the default values shown above, which are appropriate for 80-character-wide terminals.

The third operator is `=`, the assignment operator. C++ requires that the assignment operator be a member function. Like the copy constructor (see "Constructors" on page 10-4), the assignment operator does not actually make a copy of the input array, but rather references (keeps a pointer to) the input array's data; this is an optimization made purely for efficiency, and has no effect on the semantics of assignment. If you write `A = B` and then modify `B`, the values in `A` will remain unchanged:

- `mwArray &operator=(const mwArray&);`

Array Size

In MATLAB, the `size()` function returns the size of an array as an array. The MATLAB C++ Math Library provides a corresponding version of `size()` that also returns an array. Because this C++ version allocates an array to hold just two integers, it is not efficient. The `mwArray Size` member functions below return the size of an array more efficiently.

An array (a matrix is a special case) has two sizes: the number of its dimensions (for matrices, always two) and the actual size of each dimension. You can use these `Size()` functions to determine both the number of dimensions and the size of each dimension:

- `int32 Size() const`

  Return the number of dimensions.
- `int32 Size(int32 dim) const`

  Return the size (number of elements) of the indicated dimension.
- `int32 Size(int32* dims, int maxdims=2) const`

  Determine the sizes of all the dimensions of the array and return them via the given integer array, `dims`. `maxdims` is the maximum number of dimensions the function should return. The input integer array `dims` must contain enough space to store at least `maxdims` integers. If `maxdims` is less then the number of dimensions of the `mxArray`, the last dimension returned is the product of the remaining dimensions. This function's return value is the number of dimensions of the array.

For example, this code demonstrates the difference in efficiency between one of the `mwArray Size` member functions and the nonmember function.

```
int32 dims[2];
mwArray mat = rand(4,4);
mwArray sz;

// Use one of the Size member functions.
// Requires 8 bytes to return two integers, 4 and 4. No memory is
// dynamically allocated.
mat.Size(dims);

// Use the library's size function.
// Requires dynamic memory allocation of at least 85 bytes for
```

```
// the same two integers: 10 times more space, plus the
// inefficiency of data access (via pointers).
sz = size(mat);
```

Extracting Data from an mwArray

The MATLAB C++ Math Library supports several functions that let you access the data inside an `mwArray` object. All of these functions are `mwArray` member functions. For example, if you're interacting with any of the other MATLAB external interfaces – the MATLAB C Math Library, MEX files, or the MATLAB Engine – you may occasionally need to access the data inside an `mwArray` object.

GetData( )

The most basic of the functions is `GetData()`, which returns a pointer to the array data structure. This pointer is of type `mxArray*`. The array structure is an opaque data type, one in which the field names are unknown to the user. Access functions allow you to read and write the fields of the structure.

For example, to retrieve a pointer to the C++ array of double precision floating point numbers stored in an `mxArray *`, call `mxGetPr()` (to retrieve the real part of the array) or `mxGetPi()` (to retrieve the complex part). You can combine these calls with calls to `GetData()`:

```
mwArray A = magic(17);
double *real_data = mxGetPr(A.GetData());
```

Note Be careful with the pointers that `GetData()`, `mxGetPr()`, and `mxGetPi()` return. You must never free them or assign to them because the functions return pointers to the real data stored in the `mwArray`. Freeing them will cause a memory error later on.

For more details on the `mxArray` type, see the *MATLAB Application Program Interface Guide* or the header file `<matlab>/extern/include/matrix.h`. "Accessing Online Reference Documentation" on page 1-7 describes how to access the Help Desk.

SetData( )

Paired with `GetData()` is `SetData()`, which allows you to change the array data pointed to by an `mwArray` object. Use `SetData()` with care; it allows you to fool the `mwArray` reference counting system, which will lead either to memory leaks or program crashes. For example, never set the data of one mwArray to the data returned by GetData() on another mwArray:

```
mwArray A = rand(4), B = magic(10);
B.SetData(A.GetData());   // NEVER, NEVER do this.
```

Note Unless you really know what you are doing, you should never call SetData(). Use the assignment operator or the mwArray constructors instead to set the data in an array.

ExtractScalar( ) and ExtractData( )

ExtractScalar() and ExtractData() provide much safer, though somewhat slower, access to the raw data in an mwArray object. Two versions of ExtractScalar() pull a single scalar from a real or complex array. Three versions of ExtractData() copy the array data into the C++ arrays that you supply.

In the example below, A is an 11-by-11 magic square with a correspondingly sized random complex component. The numerical arguments to ExtractScalar() indicate which scalar to extract; cdata is passed by reference so that it can be modified.

```
mwArray A = magic(11) + (rand(11) * i());

double rdata, cdata;
rdata = A.ExtractScalar(9);              // Real part only
rdata = A.ExtractScalar(cdata, 17);     // Real and complex part

int32 *integers = new int32[ 11 * 11 ];
A.ExtractData(integers); // Cast doubles to integers

double *real_data = new double [ 11 * 11 ];
double *complex_data = new double [ 11 * 11 ];
A.ExtractData(real_data);                // Real part only
A.ExtractData(real_data, complex_data);  // Real and complex part
```

ExtractScalar() treats M-by-N arrays as 1-by-(M*N) vectors. A.ExtractScalar(9) is the first element in the ninth row, or alternatively, the

9th element in the first column; A.ExtractScalar(cdata, 17) is the second element in the sixth row, or alternatively the sixth element in the second column. The two ExtractScalar() functions count down the columns, wrapping from the bottom of the Nth column to the top of the (N+1)th column.

ToString( )

To extract a string from an mwArray, you can use the mwArray member function ToString(). For example,

```
mwArray A = "MATLAB";
mwString s = A.ToString();
char *c = strdup((char *)s);
```

The mwString class contains a dynamically allocated string and handles its memory management, including freeing the string when the mwString object goes out of scope. The mwString class has a cast operator that converts it to a char *.

You can safely use ToString() to construct char * function arguments, for example, strcat(str, A.ToString()). If you need to refer to the string in a context beyond the scope of the mwString object, use strdup() to make a copy of the string for yourself. Don't forget to free the copy when you're done with it.

Note Casting an mwString to a char * does not make a copy of the string. This pointer will be freed when the mwString itself goes out of scope. Do not free it yourself.

Library Routines

Introduction

This section is a reference guide for the operators that you use with arrays and the more than 400 functions contained in the MATLAB C++ Math Library.

The chapter consists of four sections:

- Operators
- MATLAB Functions
- Utility Functions
- Array Access Functions

The tables that categorize the functions include a short description of each function. Refer to the online *MATLAB C++ Math Library Reference* for a complete definition of the function syntax and arguments.

Operators

The majority of operators in the MATLAB C++ Math Library fall into two groups: the arithmetic operators that perform arithmetic on their operands and the relational operators that perform logical operations on their operands. Both types of operators return an array of results.

Arithmetic operators operate either in an element-wise fashion, like + (addition), or in an operator-dependent manner, like * (matrix multiplication). Relational operators, on the other hand, always perform an element-by-element comparison of their operands. Each element in the returned array is the result of applying the operation to the corresponding elements of the operand array. For example, if `A`, `B`, and `C` are matrices, and `C = A < B`, then `C[i] = (A[i] < B[i])`.

All operators, including a third group of miscellaneous operators, expect `mwArray` objects as operands. If you use scalars, you call the standard C++ operators. `4 + 5`, for example, does not use the matrix addition operator.

Arithmetic Operators

These binary operators perform arithmetic on their operands. The two operands for an element-wise arithmetic operator must be the same size. Operators that are not element-wise are not so uniform; they may have other operator-specific restrictions on operand size.

Table 11-1: C++ Arithmetic Operators

C++ Operator	Definition	Equivalent C++ Function
`+`	Element-wise addition	`plus()`
`-`	Element-wise subtraction	`minus(), unaryminus()`
`*`	Matrix multiplication	`mtimes()`
`/`	Matrix right division	`mrdivide()`
`^`	Matrix exponentiation	`mpower()`

Because the MATLAB syntax differs from the C++ syntax, several MATLAB operators are available in C++ as functions rather than as operators.

Table 11-2: C++ Functional Equivalents to MATLAB Operators

MATLAB Operator only	Definition	Equivalent C++ Function
`\`	Matrix left-division	`mldivide()`
`.\`	Element-wise left-division	`ldivide()`
`./`	Element-wise right-division	`rdivide()`
`.*`	Element-wise multiplication	`times()`
`.^`	Element-wise exponentiation	`power()`
`'`	Complex-conjugate transpose	`ctranspose()`
`.'`	Noncomplex transpose	`transpose()`

Relational Operators

The relational operators compare two arrays and return an identically sized array of 1's and 0's with the logical flag set. They perform an element-wise comparison of their inputs. The operators work as follows: given an expression `C = (A op B)`, where `op` is one of the operators below, then `C[i] == 1` if `(A[i] op B[i])` is true and `C[i] == 0` otherwise.

For example, if `A` is the matrix `[ 1 2 ; 3 4 ]` and `B` is the matrix `[ 0 2 ; 1 6 ]`, then `A > B` is `[ 1 0 ; 1 0 ]`. The result contains 1's where the greater-than relationship between the corresponding elements of `A` and `B` is true, and the result contains 0's where it is false. The result of a relational operation is a logical array.

"Using Logical Subscripts" in Chapter 4 provides information on logical indexing.

Table 11-3: C++ Relational Operators

C++ Operator	Definition	Equivalent C++ Function
>	Greater than	gt()
<	Less than	lt()
>=	Greater than or equal	ge()
<=	Less than or equal	le()
==	Strictly equal	eq()
!=	Not equal	neq(), ne()

Miscellaneous Operators

These operators are divided into three groups: indexing, logical, and stream. The stream operators are the only operators that do not return an array. In accordance with general practice in C++, the stream operators return their stream operand.

Table 11-4: C++ Miscellaneous Operators

C++ Operator	Definition	Equivalent C++ Function
(x)	One-dimensional indexing	Not applicable
(x, y)	Two-dimensional indexing	Not applicable
|	Logical OR	or_func()
&	Logical AND	and_func()
~	Logical NOT	not_func()

Table 11-4: C++ Miscellaneous Operators (Continued)

C++ Operator	Definition	Equivalent C++ Function
>>	Stream extraction (input)	Not applicable
<<	Stream insertion (output)	Not applicable

MATLAB Functions

The MATLAB C++ Math Library contains more than 400 functions, broadly divided into two groups: MATLAB functions, or functions that have equivalents in interpreted MATLAB; and utility functions, or functions that are necessary because of the absence of the interpreted MATLAB environment. The great majority of the functions fall into the first category, MATLAB functions. This section describes the MATLAB functions.

Each MATLAB function in the MATLAB C++ Math Library is identical to its counterpart in interpreted MATLAB. A brief description accompanies each function listed in the tables below. For additional information on the inputs and behavior of these functions, see the online *MATLAB C++ Math Library Reference*. "Accessing Online Reference Documentation" on page 1-7 describes how to access the Help Desk. Also refer to the section "How to Call C++ Library Functions" in Chapter 5 for more details on how to call these functions.

There are two categories of MATLAB functions:

- C++ versions of the MATLAB Built-In and MATLAB M-File functions.

  Each of the C++ built-in and M-file functions is named after its MATLAB equivalent. For example, the C++ version of the MATLAB eigenvalue function is named `eig()`.
- C++ functional versions of MATLAB operators.

  For example, the C++ version of the MATLAB matrix multiplication operator, `*`, is a function named `mtimes()`.

General Purpose Commands

Managing Variables

Function	Purpose
format	Set output format.
load	Retrieve variables from disk.
save	Save variables on disk.

Operators and Special Functions

Arithmetic Operator Functions

Function	Purpose
kron	Kronecker tensor product.
minus	Array subtraction (–).
mldivide	Matrix left division (\).
mpower	Matrix power (^).
mrdivide	Matrix right division (/).
mtimes	Matrix multiplication (*).
plus	Array addition (+).
power	Array power (.^).
rdivide	Array right division (./).
times	Array multiplication (.*).
unaryminus	Unary minus (–).

Relational Operator Functions

Function	Purpose
`eq`	Equality (==).
`ge`	Greater than or equal to (>=).
`gt`	Greater than (>).
`le`	Less than or equal to (<=).
`lt`	Less than (<).
`neq`	Inequality (~=).

Logical Operator Functions

Function	Purpose
`all`	True if all elements of vector are nonzero.
`and_func`	Logical AND (&).
`any`	True if any element of vector is nonzero.
`not_func`	Logical NOT (~).
`or_func`	Logical OR (|).
`xor_func`	Logical exclusive-or operation.

Set Operators

Function	Purpose
`intersect`	Set intersection of two vectors.
`ismember`	True for set member.
`setdiff`	Set difference.
`setxor`	Set exclusive OR.

Set Operators (Continued)

Function	Purpose
union_func	Set union.
unique	Set unique.

Special Operator Functions

Function	Purpose
colon	Colon operator (:).
ctranspose	Complex Conjugate Transpose (').
end	Indexes to the end of an array.
horzcat	Horizontal concatenation.
transpose	Noncomplex conjugate transpose (.').
vertcat	Vertical concatenation.

Logical Functions

Function	Purpose
find	Find indices of nonzero elements.
finite	Make elements finite.
ischar	True for character arrays.
isempty	True for empty array.
isfinite	True for finite elements of an array.
isieee	True for IEEE floating-point arithmetic.
isequal	True for input arrays of the same type, size, and contents.
isinf	True for infinite elements.
isletter	True for string elements that are letters of the alphabet.

Logical Functions (Continued)

Function	Purpose
islogical	True for logical arrays.
isnan	True for Not-a-Number.
isreal	True for noncomplex matrices.
isspace	True for whitespace characters in string matrices.
isstr	True for text strings.
isstudent	True for student editions of MATLAB.
isunix	True on UNIX machines.
isvms	True on computers running DEC’s VMS.
logical	Convert numeric values to logical.
tobool	Convert an array to a Boolean value by reducing the rank of the array to a scalar.

Bitwise Functions

Function	Purpose
bitand_func	Bitwise AND.
bitcmp	Complement bits.
bitget	Get bit.
bitmax	Maximum floating-point integer.
bitor_func	Bitwise OR.
bitset	Set bit.
bitshift	Bitwise shift.
bitxor	Bitwise XOR.

MATLAB as a Programming Language

Function	Purpose
`feval`	Function evaluation.
`lasterr`	Last error message.
`mfilename`	Return the `NULL` array. M-file execution does not apply to stand-alone applications.
`nargchk`	Validate number of input arguments.
`xyzchk`	Check arguments to 3-D data routines.

Message Display

Function	Purpose
`error`	Display message and abort function.
`warning`	Display warning message.

Elementary Matrices and Matrix Manipulation

Elementary Matrices

Function	Purpose
`eye`	Identity matrix.
`linspace`	Linearly spaced vector.
`logspace`	Logarithmically spaced vector.
`meshgrid`	X and Y arrays for 3-D plots.
`ones`	Matrix of 1's.
`rand`	Uniformly distributed random numbers.
`randn`	Normally distributed random numbers.
`zeros`	Matrix of 0's.

Basic Array Information

Function	Purpose
`disp`	Display text or matrix
`isempty`	True for empty matrix.
`isequal`	True for input arrays of the same type, size, and contents.
`islogical`	True for logical arrays.
`isnumeric`	True for numeric arrays.
`length`	Length of vector.
`logical`	Convert numeric values to logical values.
`ndims`	Number of dimensions (always 2).
`size`	Size of matrix.

Matrix Manipulation

Function	Purpose
cat	Concatenate arrays.
diag	Create or extract diagonals.
fliplr	Flip matrix in the left/right direction.
flipud	Flip matrix in the up/down direction.
ipermute	Inverse of permute.
permute	Permute array dimensions.
repmat	Replicate and tile an array.
reshape	Change size.
rot90	Rotate matrix 90 degrees.
shiftdim	Shift dimensions.
tril	Extract lower triangular part.
triu	Extract upper triangular part.

Special Constants

Function	Purpose
computer	Computer type.
eps	Floating-point relative accuracy.
flops	Floating point operation count. (Not reliable in stand-alone applications.)
inf	Infinity.
nan	Not-a-Number.
pi	3.1415926535897....

Special Constants (Continued)

Function	Purpose
realmax	Largest floating-point number.
realmin	Smallest floating-point number.

Specialized Matrices

Function	Purpose
compan	Companion matrix.
hadamard	Hadamard matrix.
hankel	Hankel matrix.
hilb	Hilbert matrix.
invhilb	Inverse Hilbert matrix.
magic	Magic square.
pascal, pascal_func	Pascal matrix.
rosser	Classic symmetric eigenvalue test problem.
toeplitz	Toeplitz matrix.
vander	Vandermonde matrix.
wilkinson	Wilkinson's eigenvalue test matrix.

Elementary Math Functions

Trigonometric Functions

Function	Purpose
acos	Inverse cosine.
acosh	Inverse hyperbolic cosine.
acot	Inverse cotangent.
acoth	Inverse hyperbolic cotangent.
acsc	Inverse cosecant.
acsch	Inverse hyperbolic cosecant.
asec	Inverse secant.
asech	Inverse hyperbolic secant.
asin	Inverse sine.
asinh	Inverse hyperbolic sine.
atan	Inverse tangent.
atan2	Four quadrant inverse tangent.
atanh	Inverse hyperbolic tangent.
cos	Cosine.
cosh	Hyperbolic cosine.
cot	Cotangent.
coth	Hyperbolic cotangent.
csc	Cosecant.
csch	Hyperbolic cosecant.
sec	Secant.
sech	Hyperbolic secant.

Trigonometric Functions (Continued)

Function	Purpose
sin	Sine.
sinh	Hyperbolic sine.
tan	Tangent.
tanh	Hyperbolic tangent.

Exponential Functions

Function	Purpose
exp	Exponential.
log	Natural logarithm.
log10	Common (base 10) logarithm.
log2	Base 2 logarithm and dissect floating-point numbers.
nextpow2	Next higher power of 2.
pow2	Base 2 power and scale floating-point numbers.
reallog	Guarantee output from log is a noncomplex matrix.
reallog10	Guarantee output from log10 is a noncomplex matrix.
realpow	Guarantee output from power is a noncomplex matrix.
realsqrt	Guarantee output from sqrt is a noncomplex matrix.
sqrt	Square root.

Complex Functions

Function	Purpose
abs	Absolute value.
angle	Phase angle.

Complex Functions (Continued)

Function	Purpose
`conj`	Complex conjugate.
`cplxpair`	Sort numbers into complex conjugate pairs.
`imag`	Complex imaginary part.
`isreal`	True for noncomplex arrays.
`real`	Real part of complex array.
`unwrap`	Remove phase angle jumps across 360° boundaries.

Rounding and Remainder Functions

Function	Purpose
`ceil`	Round toward plus infinity.
`fix`	Round toward zero.
`floor`	Round toward minus infinity.
`mod`	Modulus (signed remainder after division).
`rem`	Remainder after division.
`round`	Round toward nearest integer.
`sign`	Signum function.

Specialized Math Functions

Specialized Math Functions

Function	Purpose
beta	Beta function.
betainc	Incomplete beta function.
betaln	Logarithm of beta function.
cross	Vector cross product.
ellipj	Jacobi elliptic functions.
ellipke	Complete elliptic integral.
erf	Error function.
erfc	Complementary error function.
erfcx	Scaled complementary error function.
erfinv	Inverse error function.
expint	Exponential integral function.
gamma	Gamma function.
gammainc	Incomplete gamma function.
gammaln	Logarithm of gamma function.
legendre	Legendre functions.

Number Theoretic Functions

Function	Purpose
factor	Prime factors.
gcd	Greatest common divisor.
isprime	True for prime numbers.

Number Theoretic Functions (Continued)

Function	Purpose
`lcm`	Least common multiple.
`nchoosek`	All combinations of `n` elements taken `k` at a time.
`perms`	All possible permutations.
`primes`	Generate list of prime numbers.
`rat`	Rational approximation.
`rats`	Rational output.

Coordinate System Transforms

Function	Purpose
`cart2pol`	Transform Cartesian coordinates to polar.
`cart2sph`	Transform Cartesian coordinates to spherical.
`pol2cart`	Transform polar coordinates to Cartesian.
`sph2cart`	Transform spherical coordinates to Cartesian.

Numerical Linear Algebra

Matrix Analysis

Function	Purpose
`det`	Determinant.
`norm`	Matrix or vector norm.
`normest`	Estimate the matrix 2-norm.
`null`	Orthonormal basis for the null space.
`orth`	Orthonormal basis for the range.
`rank`	Number of linearly independent rows or columns.
`rcond`	LINPACK reciprocal condition estimator.
`rref`	Reduced row echelon form.
`subspace`	Angle between two subspaces.
`trace`	Sum of diagonal elements.

Linear Equations

Function	Purpose
`chol`	Cholesky factorization.
`cond`	Condition number with respect to inversion.
`condest`	1-norm condition number estimate.
`inv`	Matrix inverse.
`lscov`	Least squares in the presence of known covariance.
`lu`	Factors from Gaussian elimination.
`nnls`	Nonnegative least-squares.

Linear Equations (Continued)

Function	Purpose
pinv	Pseudoinverse.
qr	Orthogonal-triangular decomposition.

Eigenvalues and Singular Values

Function	Purpose
condeig	Condition number with respect to eigenvalues.
eig	Eigenvalues and eigenvectors.
hess	Hessenberg form.
poly	Characteristic polynomial.
polyeig	Polynomial eigenvalue problem.
qz	Generalized eigenvalues.
schur	Schur decomposition.
svd	Singular value decomposition.

Matrix Functions

Function	Purpose
expm	Matrix exponential.
funm	Evaluate general matrix function.
logm	Matrix logarithm.
sqrtm	Matrix square root.

Factorization Utilities

Function	Purpose
`balance`	Diagonal scaling to improve eigenvalue accuracy.
`cdf2rdf`	Complex diagonal form to real block diagonal form.
`planerot`	Generate a Givens plane rotation.
`qrdelete`	Delete a column from a QR factorization.
`qrinsert`	Insert a column into a QR factorization.
`rsf2csf`	Real block diagonal form to complex diagonal form.

Data Analysis and Fourier Transform Functions

Basic Operations

Function	Purpose
`cumprod`	Cumulative product of elements.
`cumsum`	Cumulative sum of elements.
`cumtrapz`	Cumulative trapezoidal numerical integration.
`max`	Largest component.
`mean`	Average or mean value.
`median`	Median value.
`min`	Smallest component.
`prod`	Product of elements.
`sort`	Sort in ascending order.
`sortrows`	Sort rows in ascending order.
`std`	Standard deviation.
`sum`	Sum of elements.
`trapz`	Numerical integration using trapezoidal method.

Finite Differences

Function	Purpose
`del2`	Five-point discrete Laplacian.
`diff`	Difference function and approximate derivative.
`gradient`	Approximate gradient.

Correlation

Function	Purpose
corrcoef	Correlation coefficients.
cov	Covariance matrix.
subspace	Angle between two subspaces.

Filtering and Convolution

Function	Purpose
conv	Convolution and polynomial multiplication.
conv2	Two-dimensional convolution.
deconv	Deconvolution and polynomial division.
filter	One-dimensional digital filter.
filter2	Two-dimensional digital filter.

Fourier Transforms

Function	Purpose
fft	Discrete Fourier transform.
fft2	Two-dimensional discrete Fourier transform.
fftn	Multidimensional fast Fourier transform.
fftshift	Move zeroth lag to center of spectrum.
ifft	Inverse discrete Fourier transform.
ifft2	Two-dimensional inverse discrete Fourier transform.
ifftn	Inverse multidimensional fast Fourier transform.

Sound and Audio

Function	Purpose
freqspace	Frequency spacing for frequency response.
lin2mu	Convert linear signal to mu-law encoding.
mu2lin	Convert mu-law encoding to linear signal.

Polynomial and Interpolation Functions

Data Interpolation

Function	Purpose
griddata	Data gridding.
icubic	Cubic interpolation of 1-D function.
interp1	One-dimensional interpolation (1-D table lookup).
interp1q	Quick one-dimensional linear interpolation.
interp2	Two-dimensional interpolation (2-D table lookup).
interpft	One-dimensional interpolation using FFT method.

Spline Interpolation

Function	Purpose
ppval	Evaluate piecewise polynomial.
spline	Piecewise polynomial cubic spline interpolant.

Geometric Analysis

Function	Purpose
inpolygon	Detect points inside a polygonal region.
polyarea	Area of polygon.
rectint	Rectangle intersection area.

Polynomials

Function	Purpose
conv	Multiply polynomials.
deconv	Divide polynomials.
mkpp	Make piece-wise polynomial.
poly	Construct polynomial with specified roots.
polyder	Differentiate polynomial.
polyfit	Fit polynomial to data.
polyval	Evaluate polynomial.
polyvalm	Evaluate polynomial with matrix argument.
residue	Partial-fraction expansion (residues).
resi2	Residue of a repeated pole.
roots	Find polynomial roots.
unmkpp	Supply information about piecewise polynomial.

Function Functions and ODE Solvers

Optimization and Root Finding

Function	Purpose
fmin	Minimize function of one variable.
fmins	Minimize function of several variables.
foptions	Set minimization options.
fzero	Find zero of function of one variable.
optimget	Get optimization options structure parameter values.
optimset	Create or edit optimization options parameter structure.

Numerical Integration (quadrature)

Function	Purpose
dblquad	Numerically evaluate double integral.
mquad	Numerically evaluate integral, low-order method.
quad8	Numerically evaluate integral, high-order method.

Ordinary Differential Equation Solvers

Function	Purpose
ode23	Solve differential equations, low-order method.
ode45	Solve differential equations, high-order method.
ode113	Solve non-stiff differential equations, variable order method.
ode15s	Solve stiff differential equations, variable-order method.
ode23s	Solve stiff differential equations, low-order method.

ODE Option Handling

Function	Purpose
odeget	Extract properties from options structure created with odeset.
odeset	Create or alter options structure for input to ODE solvers.

Character String Functions

General

Function	Purpose
blanks	String of blanks.
char_func	Create character array (string).
deblank	Remove trailing blanks from a string.
double_func	Convert to numeric.
str2mat	Form text matrix from individual strings.

String Tests

Function	Purpose
ischar	True for character arrays.
isletter	True for elements of the string that are letters of the alphabet.
isspace	True for whitespace characters in string arrays.

String Operations

Function	Purpose
`findstr`	Find a substring within a string.
`lower`	Convert string to lower case.
`strcat`	String concatenation.
`strcmp`	Compare strings.
`strcmpi`	Compare strings ignoring case.
`strjust`	Justify a character array.
`strmatch`	Find possible matches for a string.
`strncmp`	Compare the first `n` characters of two strings.
`strncmpi`	Compare first `n` characters of strings ignoring case.
`strrep`	Replace substrings within a string.
`strtok`	Extract tokens from a string.
`strvcat`	Vertical concatenation of strings.
`upper`	Convert string to upper case.

Base Number Conversion

Function	Purpose
`base2dec`	Base to decimal number conversion.
`bin2dec`	Binary to decimal number conversion.
`dec2base`	Decimal number to base conversion.
`dec2bin`	Decimal to binary number conversion.
`dec2hex`	Decimal to hexadecimal number conversion.

Base Number Conversion (Continued)

Function	Purpose
hex2dec	IEEE hexadecimal to decimal number conversion.
hex2num	Hexadecimal to double number conversion.

String to Number Conversion

Function	Purpose
int2str	Convert integer to string.
mat2str	Convert matrix to string.
num2str	Convert number to string.
sprintf	Convert number to string under format control.
sscanf	Convert string to number under format control.
str2double	Convert string to double-precision value.
str2num	Convert string to number.

File I/O Functions

File Opening and Closing

Function	Purpose
fclose	Close file.
fopen	Open file.

File Positioning

Function	Purpose
feof	Test for end-of-file.
ferror	Inquire file I/O error status.

File Positioning (Continued)

Function	Purpose
`frewind`	Rewind file pointer to beginning of file.
`fseek`	Set file position indicator.
`ftell`	Get file position indicator.

Formatted I/O

Function	Purpose
`fgetl`	Read line from file, discard newline character.
`fgets`	Read line from file, keep newline character.
`fprintf`	Write formatted data to file.
`fscanf`	Read formatted data from file.

Binary File I/O

Function	Purpose
`fread`	Read binary data from file.
`fwrite`	Write binary data to file.

String Conversion

Function	Purpose
`sprintf`	Write formatted data to a string.
`sscanf`	Read string under format control.

File Import/Export Functions

Function	Purpose
load	Retrieve variables from disk.
save	Save variables on disk.

Data Types

Data Types

Function	Purpose
char_func	Create character array (string)
double_func	Convert to double precision.

Object Functions

Function	Purpose
classname	Return a string representing the object's class.
isa	True if object is a given class.

Time and Dates

Current Date and Time

Function	Purpose
clock_func	Wall clock.
date	Current date string.
now	Current date and time.

Basic Functions

Function	Purpose
datenum	Serial date number.
datestr	Date string format.
datevec	Date components.

Date Functions

Function	Purpose
calendar	Calendar.
eomday	End of month.
weekday	Day of the week.

Timing Functions

Function	Purpose
etime	Elapsed time function.
tic,toc	Stopwatch timer functions.

Multidimensional Array Functions

Function	Purpose
cat	Concatenate arrays.
ind2sub	Subscripts from linear index.
ipermute	Inverse permute array dimensions.
ndims	Number of array dimensions.
permute	Permute array dimensions.
shiftdim	Shift dimensions.
sub2ind	Linear index from multiple subscripts.

Cell Array Functions

Function	Purpose
cell	Create cell array.
cell2struct	Convert cell array into structure array.
celldisp	Display cell array contents.
cellfun	Apply a cell function to a cell array.
cellhcat	Horizontally concatenate cell arrays.
cellstr	Create cell array of strings from character array.
deal	Deal inputs to outputs.
iscell	True for cell array.
iscellstr	True for a cell array of strings.
num2cell	Convert numeric array into cell array.

Structure Functions

Function	Purpose
fieldnames	Get structure field names.
getfield	Get structure field contents.
isfield	True if field is in structure array.
isstruct	True for structures.
rmfield	Remove structure field.
setfield	Set structure field contents.
struct	Create or convert to structure array.
struct2cell	Convert structure array into cell array.

Sparse Matrix Functions

Elementary Sparse Matrices

Function	Purpose
spdiags	Sparse matrix formed from diagonals.
speye	Sparse identity matrix.
sprand	Sparse uniformly distributed random matrix.
sprandn	Sparse normally distributed random matrix.
sprandsym	Sparse random symmetric matrix.

Full to Sparse Conversion

Function	Purpose
find	Find indices of nonzero elements.
full	Convert sparse matrix to full matrix.

Full to Sparse Conversion (Continued)

Function	Purpose
`sparse`	Create sparse matrix.
`spconvert`	Import from sparse matrix external format.

Working with Nonzero Entries of Sparse Matrices

Function	Purpose
`issparse`	True for sparse matrix.
`nnz`	Number of nonzero matrix elements.
`nonzeros`	Nonzero matrix elements.
`nzmax`	Amount of storage allocated for nonzero matrix elements.
`spalloc`	Allocate space for sparse matrix.
`spfun`	Apply function to nonzero matrix elements.
`spones`	Replace nonzero sparse matrix elements with 1's.

Reordering Algorithms

Function	Purpose
`colmmd`	Column minimum degree permutation.
`colperm`	Column permutation.
`dmperm`	Dulmage-Mendelsohn permutation.
`randperm`	Random permutation.
`symmmd`	Symmetric minimum degree permutation.
`symrcm`	Symmetric reverse Cuthill-McKee permutation.

Linear Algebra

Function	Purpose
`cholinc`	Incomplete Cholesky factorization.
`condest`	1-norm condition number estimate.
`eigs`	A few eigenvalues.
`luinc`	Incomplete LU factorization.
`normest`	Estimate the matrix 2-norm.
`svds`	A few singular values.

Linear Equations (Iterative Methods)

Function	Purpose
`bicg`	BiConjugate Gradients Method.
`bicgstab`	BiConjugate Gradients Stabilized Method.
`cgs`	Conjugate Gradients Squared Method.
`gmres`	Generalized Minimum Residual Method.
`pcg`	Preconditioned Conjugate Gradients Method.
`qmr`	Quasi-Minimal Residual Method.

Miscellaneous

Function	Purpose
`spaugment`	Form least squares augmented system.
`spparms`	Set parameters for sparse matrix routines.
`symbfact`	Symbolic factorization analysis.

Utility Functions

In addition to its mathematical functions, the interpreted MATLAB environment provides services such as memory management and array input and output. The MATLAB C++ Math Library cannot draw on the MATLAB environment for these essential services, so it provides its own services that initialize and control the library environment and that help you perform indexing.

These functions require several new types that describe pointers to functions. You will find these types used in the tables of functions below; these types are not part of MATLAB.

```
// Used for print handling functions
typedef void (*mwOutputFunc)(const char *);

// Used for error handling functions
typedef void (*mwErrorFunc)(const char*, mwBool);

// Used for exception handling
typedef void (*mwExceptionMsgFunc)(const mwException &);

// Used for memory allocation functions
typedef void *(*mwMemAllocFunc)(size_t);
typedef void (*smwMemFreeFunc)(void *);
typedef void *(*mwMemReallocFunc)(void *, size_t);
typedef void *(*mwMemCallocFunc)(size_t, size_t);
```

For more information on the error and exception handling functions, refer to the section “Handling Exceptions” in Chapter 7; for more information on print handling, see “Defining a Print Handler” in Chapter 7.

Print Handling

Function	Purpose
`mwOutputFunc mwGetPrintHandler(void);`	Return a pointer to the function specified in the most recent call to `mwSetPrintHandler()` or to the default print handler, if you haven’t specified a print handler.
`void mwSetPrintHandler(mwOutputFunc f);`	Set the print handling routine. The print handler is responsible for handling all “normal” (non-error) output.

Error and Exception Handling

Function	Purpose
`void mwDisplayException(const mwException &ex);`	Using the error handler, displays the given exception.
`mwErrorFunc mwGetErrorMsgHandler(void);`	Return a pointer to the function specified in the most recent call to `mwSetErrorMsgHandler()` or to the default error handler, if you haven’t specified an error handler.
`mwExceptionMsgFunc mwGetExceptionMsgHandler(void);`	Return a pointer to the function specified in the most recent call to `mwSetExceptionMsgHandler()` or to the default exception message handler, if you haven’t specified an exception message handler.

Error and Exception Handling (Continued)

Function	Purpose
`void mwSetErrorMsgHandler(mwErrorFunc f);`	Set the error handling routine. The error handler is responsible for handling all error message output.
`void mwSetExceptionMsgHandler(mwExceptionMsgFunc f)`	The default exception handling function simply prints the exception using the error handling routine. If this behavior is inappropriate for your application, this function allows you to set an alternate exception handling function.

Memory Allocation

Function	Purpose
`void mwSetLibraryAllocFcns( mwMemCallocFunc callocProc, mwMemFreeFunc freeProc, mwMemReallocFunc reallocProc, mwMemAllocFunc mallocproc, mwMemCompactFunc=0);`	Set the MATLAB C++ Math Library's memory management functions. Gives you complete control over memory management.

MATLAB uses the `:` (colon) operator to generate sequences of numbers: both vectors and matrix indices. Because the colon operator is unavailable in C++, the MATLAB C++ Math Library provides two families of functions, `ramp()` and `colon()`, to support the same functionality. The `ramp()` functions are best suited for generating vectors, the `colon()` functions for array indices.

Chapter 4, “Indexing into Arrays,” contains more details on the use of generated sequences in array indexing operations.

Generating Sequences

Function	Purpose
`mwArray ramp(mwArray start, mwArray end);`	Generate a vector of `(end-start)+1` elements. The elements in the vector are `start`, `start+1`, `start+2`, `. . .` , `start+n`, `end`. Each element in the vector is one greater than the preceding element, with the possible exception of the last element (see below).
`mwArray ramp(mwArray start, mwArray step, mwArray end);`	Generate a vector of `((end-start)/step)+1` elements. The elements in the vector are `start`, `start+step`, `start+(2*step)`, `start+(3*step)`, `...`,`start+(n*step)`, `end`. Each element in the vector is `step` greater than the preceding element, with the possible exception of the last element. Iteration stops when `start+(n*step)` is larger than `end`, yet the last value in the vector is always `end`; this can decrease the distance between the last two elements to less than `step`. Specifying a negative step generates a decreasing sequence; specifying a sequence that will not terminate raises an exception.

Indexing

Function	Purpose
`mwIndex colon();`	Generate a “sequence” of indices. `colon()` stands for “every value.” For example, `A(colon())` means every value in the matrix. `A(1,colon())` means every column in the first row.
`mwIndex colon(mwArray start, mwArray end);`	This function is identical to the analogous `ramp()` function, except that it is more efficient when used as a matrix index.

Indexing (Continued)

Function	Purpose
`mwIndex colon(mwArray start, mwArray step, mwArray stop);`	This function is identical to the analogous `ramp()` function, except that it is more efficient when used as an array index.
`mwArray end(mwArray &mat, mwArray &x, mwArray &y);`	Generate the last index for an array dimension. Acts like `end` in the MATLAB expression `A(3,6:end)`. `x` is the dimension to compute end for. Use 1 to indicate the row dimension; use 2 to indicate the column dimension. `y` is the number of indices in the subscript.

Array Access Functions

The Array Access and Creation Library contains the array creation and access routines for the `mxArray` data type. In general, the arguments to these functions are `mxArray*` pointers instead of `mwArray` variables. For example, `mxCreateDoubleMatrix()` creates an `mxArray`; `mxDestroyArray()` destroys one.

Refer to the online *MATLAB Application Program Interface Reference* for a detailed definition of each function. The *MATLAB Application Program Interface Guide* also documents these functions.

Note You can recognize an Array Access and Creation Library routine by its prefix `mx`.

Array Access Routines

Function	Purpose
`mxCalloc, mxFree`	Allocate and free dynamic memory using MATLAB's memory manager.
`mxClearLogical`	Clear the logical flag.
`mxCreateCellArray`	Create an unpopulated N-dimensional cell `mxArray`.
`mxCreateCellMatrix`	Create an unpopulated 2-D cell `mxArray`.
`mxCreateCharArray`	Create an unpopulated N-dimensional string `mxArray`.
`mxCreateCharMatrixFromStrings`	Create a populated 2-D string `mxArray`.
`mxCreateDoubleMatrix`	Create an unpopulated 2-D, double-precision, floating-point `mxArray`.
`mxCreateNumericArray`	Create an unpopulated N-dimensional numeric `mxArray`.
`mxCreateSparse`	Create a 2-D unpopulated sparse `mxArray`.

Array Access Routines (Continued)

Function	Purpose
`mxCreateString`	Create a 1-by-n string `mxArray` initialized to the specified string.
`mxCreateStructArray`	Create an unpopulated N-dimensional structure `mxArray`.
`mxCreateStructMatrix`	Create an unpopulated 2-D structure `mxArray`.
`mxDestroyArray`	Free dynamic memory allocated by an `mxCreate` routine.
`mxDuplicateArray`	Make a deep copy of an array.
`mxGetCell`	Get a cell's contents.
`mxGetClassID`	Get (as an enumerated constant) an `mxArray`'s class.
`mxGetClassName`	Get (as a string) an `mxArray`'s class.
`mxGetData`	Get pointer to data.
`mxGetDimensions`	Get a pointer to the dimensions array.
`mxGetElementSize`	Get the number of bytes required to store each data element.
`mxGetEps`	Get value of eps.
`mxGetField`	Get a field value, given a field name and an index in a structure array.
`mxGetFieldByNumber`	Get a field value, given a field number and an index in a structure array.
`mxGetFieldNameByNumber`	Get a field name, given a field number in a structure array.
`mxGetFieldNumber`	Get a field number, given a field name in a structure array.
`mxGetImagData`	Get pointer to imaginary data of an `mxArray`.
`mxGetInf`	Get the value of infinity.

Array Access Routines (Continued)

Function	Purpose
mxGetIr	Get the ir array of a sparse matrix.
mxGetJc	Get the jc array of a sparse matrix.
mxGetM, mxGetN	Get the number of rows (M) and columns (N) of an array.
mxGetName, mxSetName	Get and set the name of an mxArray.
mxGetNaN	Get the value of Not-a-Number.
mxGetNumberOfDimensions	Get the number of dimensions.
mxGetNumberOfElements	Get number of elements in an array.
mxGetNumberOfFields	Get the number of fields in a structure mxArray.
mxGetNzmax	Get the number of elements in the ir, pr, and (if it exists) pi arrays.
mxGetPi, mxGetPr	Get the real and imaginary parts of an mxArray.
mxGetScalar	Get the real component from the first data element of an mxArray.
mxGetString	Copy the data from a string mxArray into a C-style string.
mxIsChar	True for a character array.
mxIsClass	True if mxArray is a member of the specified class.
mxIsComplex	True if data is complex.
mxIsDouble	True if mxArray represents its data as double-precision, floating-point numbers.
mxIsEmpty	True if mxArray is empty.
mxIsFinite	True if value is finite.
mxIsInf	True if value is infinite.
mxIsLogical	True if mxArray is Boolean.

Array Access Routines (Continued)

Function	Purpose
mxIsNaN	True if value is Not-a-Number.
mxIsNumeric	True if mxArray is numeric or a string.
mxIsSingle	True if mxArray represents its data as single-precision, floating-point numbers.
mxIsSparse	Inquire if an mxArray is sparse. Always false for the MATLAB C Math Library.
mxIsStruct	True if a structure mxArray.
mxMalloc	Allocate dynamic memory using MATLAB's memory manager.
mxRealloc	Reallocate memory.
mxSetCell	Set the value of one cell.
mxSetData	Set pointer to data.
mxSetDimensions	Modify the number of dimensions and/or the size of each dimension.
mxSetField	Set a field value of a structure array, given a field name and an index.
mxSetFieldByNumber	Set a field value in a structure array, given a field number and an index.
mxSetImagData	Set imaginary data pointer for an mxArray.
mxSetIr	Set the ir array of a sparse mxArray.
mxSetJc	Set the jc array of a sparse mxArray.
mxSetLogical	Set the logical flag.
mxSetM, mxSetN	Set the number of rows (M) and columns (N) of an array.

Array Access Routines (Continued)

Function	Purpose
mxSetNzmax	Set the storage space for nonzero elements.
mxSetPi, mxSetPr	Set the real and imaginary parts of an mxArray.

A

Directory Organization

Introduction

This section describes the directory organization of the MATLAB C++ Math Library on UNIX and Microsoft Windows systems.

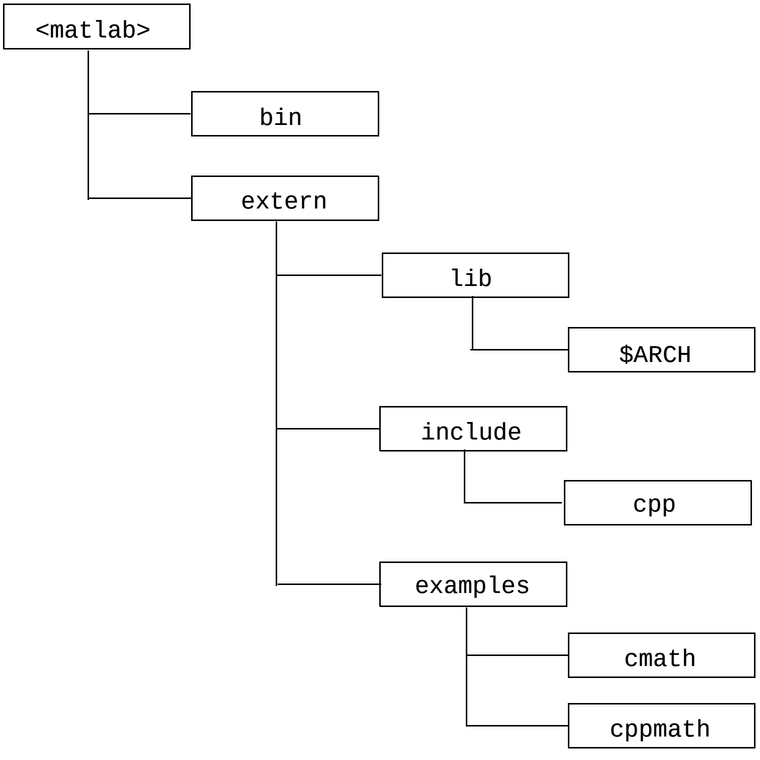
Refer to this section to find out what files the MATLAB C++ Math Library installs on your computer, what the purpose of each file is, and where each file is stored. For instructions on how to install the software, see “Installing the C++ Math Library” in Chapter 1.

The MATLAB C++ Math Library is part of a family of tools offered by The MathWorks. All MathWorks products are stored under a single directory, the MATLAB root directory. Separate directories for the major product categories are located under the MATLAB root.

The MATLAB C++ Math Library is installed in the `extern` directory, where products external to MATLAB are installed, and in the `bin` directory. If you have other MathWorks products, there are other directories directly below the MATLAB root.

Directory Organization on UNIX

This figure illustrates the directory structure for the MATLAB C++ Math Library files. `<matlab>` represents the top-level directory where MATLAB is installed on your system. `$ARCH` specifies a particular UNIX platform.



<matlab>/bin

The `<matlab>/bin` directory contains the `mbuild` script and the scripts it uses to build your code..

`mbuild`	Shell script that controls the building and linking of your code.
`mbuildopts.sh`	Options file that controls the switches and options for your C compiler. It is architecture specific. When you execute `mbuild –setup`, this file is copied to your home directory.

<matlab>/extern/lib/$ARCH

The `<matlab>/extern/lib/$ARCH` directory contains the MATLAB C++ Math libraries, where `$ARCH` specifies a particular UNIX platform. For example, on a Sun SPARCstation running Solaris, `sol2` is the name of the `$ARCH` directory.

`libmat.`*ext*	MAT-file access routines to support `mlfLoad` and `mlfSave`.
`libmatlb.`*ext*	MATLAB Built-In Math Library. Contains stand-alone versions of MATLAB built-in math functions and operators. Required for building stand-alone applications.
`libmatpp.`*ext*	MATLAB C++ Math Library. Contains the C++ interface to the Built-In and M-File library routines. Required for building stand-alone C++ applications.
`libmi.`*ext*	Internal MAT-file access routines.
`libmmfile.`*ext*	MATLAB M-File Math Library. Contains stand-alone versions of the MATLAB math M-files. Needed for building stand-alone applications that require MATLAB M-file math functions.

libmx.*ext*	MATLAB Array Access and Creation Library. Contains array access routines.
libut.*ext*	MATLAB Utilities Library. Contains the utility routines used by various components.

In the listing above, .*ext* is .a on IBM RS/6000; .so on Solaris, Alpha, Linux, and SGI; and .sl on HP 700.

<matlab>/extern/include

The <matlab>/extern/include directory contains the C header files for developing stand-alone applications. Because the MATLAB C++ Math Library contains the MATLAB C Math Library, the header file matlab.h file is required.

libmatlb.h	Header file containing the prototypes for the MATLAB Built-In Math Library functions.
libmmfile.h	Header file containing the prototypes for the MATLAB M-File Math Library functions.
matlab.h	Header file for the MATLAB C Math Library.
matrix.h	Header file containing the definition of the mxArray type and function prototypes for array access routines.

<matlab>/extern/include/cpp

The <matlab>/extern/include/cpp directory contains the C++ header files for developing stand-alone C++ applications.

matlab.hpp	Header file for the MATLAB C++ Math Library.
version.h	Architecture specific C++ compiler definitions.
mathwork.h	Declaration of scalar types.

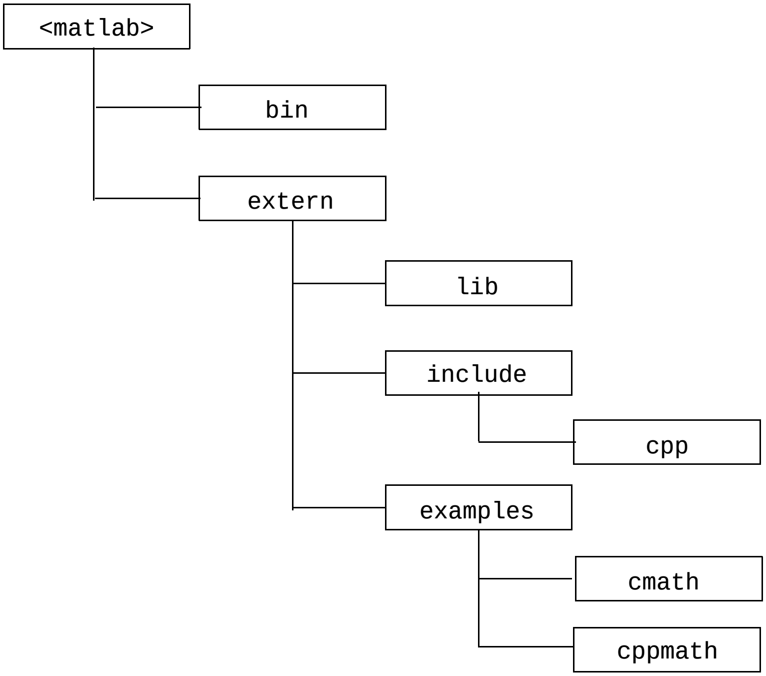
<matlab>/extern/examples/cppmath

The `<matlab>/extern/examples/cppmath` directory holds the sample C++ programs presented in this book.

`ex1.cpp`	The source code for "Example Program: Creating Arrays and Array I/O (ex1.cpp)" on page 3-15.
`ex2.cpp`	The source code for "Example Program: Calling Library Functions (ex2.cpp)" on page 5-12.
`ex3.cpp`	The source code for "Example – Passing Functions As Arguments (ex3.cpp)" on page 5-19.
`ex4.cpp`	The source code for "Example Program: Writing Simple Functions (ex4.cpp)" on page 2-19.
`ex5.cpp`	The source code for "Example Program: Handling Exceptions (ex5.cpp)" on page 7-9.
`ex6.cpp`	The source code for "Example – Using File I/O Functions (ex6.cpp)" on page 8-15.
`ex7.cpp`	The source code for "Example – Using load() and save() (ex7.cpp)" on page 8-22.
`ex8.cpp`	The source code for "Example Program: Rewriting roots.m in C++ (ex8.cpp)" on page 9-11.
`release.txt`	Release notes for the current release of the MATLAB C++ Math Library.

Directory Organization on Microsoft Windows

This figure illustrates the folders that contain the MATLAB C++ Math Library files. `<matlab>` represents the top-level folder where MATLAB is installed on your system.



<matlab>\bin

The <matlab>\bin directory contains the Dynamic Link Libraries (DLLs) required by stand-alone applications, and the batch file mbuild, which controls the build and link process for you. <matlab>\bin must be on your path for your applications to run. All DLLs are in WIN32 format.

libmat.dll	MAT-file access routines to support mlfLoad() and mlfSave().
libmatlb.dll	MATLAB Built-In Math Library. Contains stand-alone versions of MATLAB built-in math functions and operators. Required for building stand-alone applications.
libmi.dll	Internal MAT-file access routines.
libmmfile.dll	MATLAB M-File Math Library. Contains stand-alone versions of the MATLAB math M-files. Needed for building stand-alone applications that require MATLAB M-file math functions.
libmx.dll	MATLAB Array Access and Creation Library. Contains array access routines.
libut.dll	MATLAB Utilities Library. Contains the utility routines used by various components.
mbuild.bat	Batch file that helps you build and link stand-alone executables.
compopts.bat	Default options file for use with mbuild.bat. Created by mbuild –setup.
Options files for mbuild.bat	Options and settings for the C++ compiler to create stand-alone applications, e.g., msvccompp.bat for use with Microsoft Visual C/C++.

<matlab>\extern\lib

The <matlab>\extern\lib directory contains compiler-specific libraries. Because different linkers use different file formats, we provide versions of the

MATLAB C++ Math Library for each compiler we support: Borland, Microsoft Visual C++, and Watcom.

Each library contains the C++ interface to the MATLAB C Built-In and M-File Math libraries. The MATLAB C++ Math Library is required for building stand-alone C++ applications. These libraries are static libraries.

`libmatpb50.lib`
`libmatpb52.lib`
`libmatpb53.lib`
	MATLAB C++ Math Library for the Borland C++ compiler, v5.0, v5.2, and v5.3.

`libmatpm.lib`	MATLAB C++ Math Library for the Microsoft Visual C++ compiler.
`libmatpw106.lib` `libmatpw11.lib`	MATLAB C++ Math Library for the Watcom C++ compiler, v10.6 and v11.

<matlab>\extern\include

The `<matlab>\extern\include` directory contains the C header files for developing stand-alone C++ applications. Because the MATLAB C++ Math Library contains the MATLAB C Math Library, the header files `matlab.h` and `matrix.h` are required by the C++ library.

The listed `.def` files are used by the Microsoft Visual C++ and Borland compilers. The `lib*.def` files are used by MSVC++ and the `_lib*.def` files are used by Borland.

`libmatlb.h`	Header file containing the prototypes for the MATLAB Built-In Math Library functions.
`libmmfile.h`	Header file containing the prototypes for the MATLAB M-File Math Library functions.
`matlab.h`	Header file for the MATLAB C Math Library.
`matrix.h`	Header file containing the definition of the `mxArray` type and function prototypes for array access routines.
`libmat.def` `_libmat.def`	Contains names of functions exported from the MAT-file DLL.

libmatlb.def _libmatlb.def	Contains names of functions exported from the MATLAB Built-In Math Library DLL.
libmmfile.def _libmmfile.def	Contains names of functions exported from the MATLAB M-File Math Library DLL.
libmx.def _libmx.def	Contains names of functions exported from libmx.dll.

<matlab>\extern\include\cpp

The <matlab>\extern\include\cpp directory contains the C++ header files for developing stand-alone C++ applications.

matlab.hpp	Header file for the MATLAB C++ Math Library.
version.h	Architecture specific C++ compiler definitions.
mathwork.h	Declaration of scalar types.

<matlab>\extern\examples\cppmath

The <matlab>\extern\examples\cppmath directory contains the sample C++ examples that are presented in this book.

ex1.cpp	The source code for "Example Program: Creating Arrays and Array I/O (ex1.cpp)" on page 3-15.
ex2.cpp	The source code for "Example Program: Calling Library Functions (ex2.cpp)" on page 5-12.
ex3.cpp	The source code for "Example – Passing Functions As Arguments (ex3.cpp)" on page 5-19.
ex4.cpp	The source code for "Example Program: Writing Simple Functions (ex4.cpp)" on page 2-19.
ex5.cpp	The source code for "Example Program: Handling Exceptions (ex5.cpp)" on page 7-9.

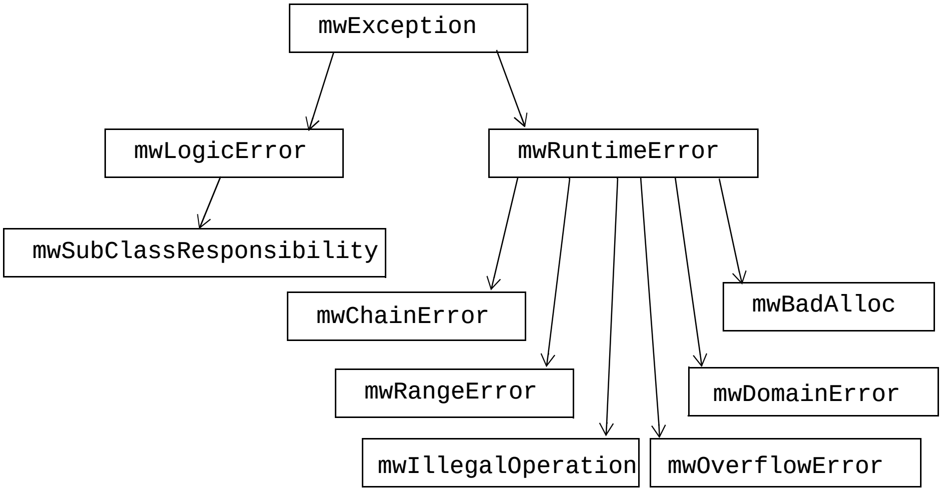
ex6.cpp	The source code for “Example – Using File I/O Functions (ex6.cpp)” on page 8-15.
ex7.cpp	The source code for “Example – Using load() and save() (ex7.cpp)” on page 8-22.
ex8.cpp	The source code for “Example Program: Rewriting roots.m in C++ (ex8.cpp)” on page 9-11.
release.txt	Release notes for the current release of the MATLAB C++ Math Library.

B

Exception Classes

Overview

The MATLAB C++ Math Library defines a hierarchy of 10 exception classes with `mwException` as the base class. The root class, `mwException`, has two children: `mwLogicError` and `mwRuntimeError`. Most of the exception classes are children of `mwRuntimeError`. The following figure illustrates this hierarchy.



You can make use of these exception classes in your own code. You may even derive further exception classes from the ones presented here. For examples of how to derive a class from `mwException` or one of its subclasses, see the file `stdexcpt.h` in the `extern/include/cpp` directory of your installation.

Exception Class Descriptions

Each exception class is described briefly below.

`mwException`

The base class for the exception classes, `mwException`, has two direct descendants: `mwLogicError` and `mwRuntimeError`. Most `catch`-blocks catch `mwException` objects rather than instances of one of its subclasses.

`mwLogicError`

A subclass of `mwException`. Logic errors occur as the result of bugs, either in your code or in the library itself. Generally, they are fatal, which means no corrective action can be taken by the program. The code itself needs to be modified.

`mwSubclassResponsibility`

A subclass of `mwLogicError`. Exceptions of this type are thrown when a subclass does not completely reimplement the virtual interface of its parent class. Under normal circumstances, you should never see an exception of this type. Refer to a C++ reference guide for a more thorough treatment of virtual interfaces and inheritance.

`mwRuntimeError`

A subclass of `mwException`. Most commonly encountered errors fall into this category. Run-time errors are often nonfatal and indicate nothing more serious than invalid input or resource conflicts. Some, however, can be fatal.

`mwChainError`

A subclass of `mwRuntimeError`. An `mwChainError` is used to wrap up and rethrow exceptions that were caught but not completely handled by a `catch`-block.

`mwRangeError`

A subclass of `mwRuntimeError`. An unrepresentable or unexpected value has resulted from a computation. This error may point to a problem in either the input or the code for the computation.

`mwDomainError`

A subclass of `mwRuntimeError`. A function has encountered unexpected or corrupt input. Of all the error classes listed here, domain errors are the easiest from which to recover.

`mwOverflowError`

A subclass of `mwRuntimeError`. Some form of arithmetic overflow has occurred.

`mwIllegalOperation`

A subclass of `mwRuntimeError`. The programmer has attempted to perform an operation that is not supported. Often, this error occurs when an operation is not yet implemented. The only recourse for errors of this type is to avoid the offending operation.

`mwBadAlloc`

A subclass of `mwRuntimeError`. The operating system has denied the program's request for more dynamic memory, generally resulting in a `"A memory allocation request failed"` message.

Error Messages

Overview

This section provides an alphabetical list of the error messages issued by the MATLAB C++ Math Library. Accompanying each error message is a short description explaining why the error occurred and, where applicable, what can be done to correct it.

Error Types

You may encounter errors other than those listed below. In all likelihood those errors come from the mathematical code that forms the foundation for the MATLAB C++ Math Library. For the most part, the messages are self-explanatory.

Many of the errors listed are internal errors. Internal errors occur when the library fails a built-in consistency check. Internal errors can be caused by a bug in the library or in your program. For instance, your program may have written randomly into memory and destroyed some library data. If you have access to a memory usage verification program like Purify or BoundsChecker, try running it on your program. If you are reasonably sure that the error is not caused by a bug in your program, please report internal errors to The MathWorks.

Reporting Errors

When reporting an error to The MathWorks, please be as specific as possible. Include a small example that replicates the problem along with any instructions needed to compile and run the example. You may report bugs through any of our normal support channels. Electronic mail is the most efficient way to contact us; our support address is: `support@mathworks.com`.

Alphabetized Error Messages

`Cannot extract shared data. Use copy() first.`

The function `mwArray::FreezeData()` issues this error when it is called on an array with a reference count higher than one. See "Example – Passing Functions As Arguments (ex3.cpp)" in Chapter 5 for more details on using `FreezeData()`.

`Deleting Matrix with nonzero reference count = <number>.`

A matrix that is still in use is being deleted. This is an internal error indicating the matrix reference counting code has become confused.

`Don't set library allocation functions to NULL.`

The library memory allocation and deallocation functions must never be `NULL`. This error indicates a user's attempt to set one of the library's allocation functions to `NULL`, for example, `mwSetFreeHandler(NULL)`.

`Don't set library error handler to NULL.`

By default, the error handler throws an exception. You can change this behavior, but you must always have an error handling function. If you try to set the error handler to `NULL`, for example, `mwSetErrorHandler(NULL)`, you'll get this message.

`Don't set library error message handler to NULL.`

The library error message handling function must never be `NULL`. This error indicates a user's attempt to set it to `NULL`, for example, `mwSetErrorMsgHandler(NULL)`.

`Don't set library exception message handler to NULL.`

The library exception message handling function must never be `NULL`. This error indicates a user's attempt to set it to `NULL`, for example, `mwSetExceptionMsgHandler(NULL)`.

`Don't set library print handler to NULL.`

The library print handling function must never be `NULL`. This error indicates a user attempt to set it to `NULL`, for example, `mwSetPrintHandler(NULL)`.

`Extraction from NULL matrix.`

This internal error occurs when the program attempts to extract a double-precision floating-point number from a 1-by-1 matrix that contains no data.

`Inconsistent precision: expecting <number>, found <number>.`

This is an internal error issued by the matrix printing routine when it discovers that an element of a matrix is too large (too many digits) to print in the space allocated for it.

`Input to <name> must be 1-by-2; was <number>-by-<number>.`

The matrix creation functions, `ones()`, `eye()`, `zeros()`, `magic()`, `rand()`, and `randn()`, accept one or two doubles or a matrix of two doubles as input arguments. This error occurs when the input matrix is not 1-by-2, for example, `ones(zeros(4))`. The inner call, to `zeros()`, succeeds and returns a 4-by-4 matrix. The second call, to `ones()` with a 4-by-4 matrix, produces this error.

`Line width must be positive: <number> isn't.`

You can set the width of the lines (the maximum number of characters that will fit on a line) on your display screen; the wider the screen, the more matrix elements will be displayed on each line. However, if you specify a negative or zero width, you will see this error.

`Matrix input format error: All rows must be the same length (<number>).`

All the rows in a matrix must contain the same number of columns. This error occurs when the matrix input routine, `operator>>()`, detects a “ragged” matrix; i.e., one in which all the rows do not contain the same number of columns. For example, `[ 1 2 ; 3 ; 4 5 ]`. The second row contains only one column, while rows one and three contain two columns.

`Matrix input format error: Can't find scale factor.`

A scale factor, for example, `1e–10 * [1 2; 3 4]`, may precede a matrix in the input stream. This error occurs when the first nonblank character read by the matrix input routine is neither a bracket `[` nor the beginning of a valid double-precision floating-point number. A scale factor of `0.0` also causes this error.

`Matrix input format error: Complex numbers must end with an 'i'. Found '<character>' instead.`

3+5i, 0-2i, and even 9.35i are all valid complex numbers. The terminating 'i' character indicates the numbers are complex. This error occurs when the input routine thinks it is reading a complex number and is surprised to find that the number being read does not end with an 'i'. Missing whitespace between columns, for example `[1–2; 3 4]`, causes this error. Whitespace inserted between the `1` and the `–`, `[1 –2; 3 4]`, makes this into a valid matrix.

`Matrix input format error: Expecting a digit, found '<character>'.`

When the characters + and – occur in the input stream, they must be followed by a digit between `0` and `9`. This error occurs when the input routine encounters a + or – and the next (nonwhitespace) character is not a digit between `0` and `9`.

`Matrix input format error: Missing '*' from scale factor.`

A matrix scale factor consists of a nonzero double-precision floating-point number followed by an asterisk denoting multiplication. If the asterisk is not present, this error occurs.

`Matrix input format error: Missing '['.`

The matrix input format stipulates that all matrices begin with a bracket `[`. This error occurs when the matrix input routine, `operator>>()`, can't find the initial bracket `[` character.

`Matrix input format error: Missing ']'.`

The matrix input format stipulates that all matrices (except string matrices) end with a bracket `]`. This error occurs when the matrix input routine, `operator>>()`, can't find the terminating bracket `]` character.

`Matrix input format error: String matrix terminated with <character> rather than '.`

To be recognized as a string matrix, a matrix must begin and end with a single quote character. This error occurs when the trailing single quote character is missing.

`Matrix input format error: Unrecognized character: '<character>'.`

Only the digits 0-9, the symbols + and –, the period `.`, the semicolon `;`, the letter `e` (for scientific notation), the letter `i` (to indicate a complex number), and whitespace characters (space, tab, and carriage return) are permitted between the opening bracket `[` and closing bracket `]` of a matrix definition. This error indicates that a character outside that set appeared in the input stream. Correct this problem by removing the out-of-range character.

`A memory allocation request failed.`

The program is out of memory. There is very little you can do about this. Try to rewrite your code to use less memory. Exit any nonessential programs. Increase the size of your swap partition. Add more memory to your machine.

`Need array pointer to determine size of ':'.`

This is an internal error that indicates the `mwArray` object is corrupt.

`Not yet implemented: <some text>.`

The feature you are trying to use, indicated by `<some text>` in the message, has not yet been implemented.

`Null matrix data.`

The `mwArray` copy constructor checks the data pointer of the matrix it is copying. If the data pointer is `NULL`, this internal error occurs.

`Null matrix on left-hand side.`

Assignment with `NULL` matrices is a special case. If you see this error message, it means the library failed to detect this case correctly.

`Null reference matrix in index operation.`

Matrix index operations generally involve an intermediate `mwSubArray` object. The `mwSubArray` contains a pointer to the matrix on which the index operation was performed. This pointer should never be `NULL`. This error occurs when the pointer is `NULL`.

`Null Reference() pointer!`

The reference field of an `mwArray` object is `NULL` when it should not be. This is an internal error.

`Only 1-by-1 matrices can be cast to doubles. Matrix is <number>-by-<number>.`

This error occurs when you treat a matrix with a size other than 1-by-1 as a scalar; for example, by assigning it to a `double`. The error message displays the dimensions of the matrix. Correct this error by using an indexing operation to extract the number you want from the matrix or, if this error occurs in a logical expression (for example, an `if`-statement), by placing a call to `tobool()` around the conditional expression.

`Only noncomplex matrices can be cast to doubles.`

You cannot cast a matrix with a complex component to a double, even if the matrix is 1-by-1. If you were able to, the complex component of the matrix would be lost in translation. If you really need access to the complex component of a matrix `A`, the code `mxGetPi(A.GetData())` will return a pointer to the two-dimensional array of complex matrix data. However, by using this construct, you are circumventing the safeguards built into the library. If you write to this array, the `mwArray` object(s) that contain(s) it may no longer be able to function properly. Reading from the array is relatively safe.

`Output pointer (first arg.) NULL.`

There is a special, efficient, version of the `size()` routine that returns the size of the matrix as two integers rather than as a matrix. The first argument to this version of `size()` is a pointer to an integer; `size()` stores the number of columns there. If that pointer is `NULL`, this error occurs.

`Premature end of file.`

The matrix input routine, `operator>>()`, came to the end of the file before reading a complete matrix. Check to be sure the file exists and that the data in it is correct. Remember: `operator>>()` can only read ASCII files.

`<function> with complex result.`

Four functions, `reallog()`, `reallog10()`, `realpow()`, and `realsqrt()`, verify that their return value is noncomplex. If the return value is complex, this error occurs. This is caused by incorrect input, for example, `realsqrt(-1)`.

Index

B

C

F

N

P

Q

R

S