MATLAB®

C++ Math Library

Computation

Visualization

Programming

User's Guide

Version 1.2

How to Contact The MathWorks:

508-647-7000 Phone

508-647-7001 Fax

The MathWorks, Inc.
24 Prime Park Way
Natick, MA 01760-1500 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 1996 First printing
January 1998 Revised for MATLAB 5.2

Contents

1 Getting Ready

2 Fundamentals

3 Writing Programs

4 Using the Library

5

6 Directory Organization

A Tricks, Tips, and Techniques

B The mxArray Structure

C Error Messages

1

Getting Ready

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.

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.

The MATLAB C++ Math Library

The MATLAB C++ Math Library consists of approximately 350 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.

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.

While the library provides a great many functions, it does not contain all of MATLAB. The MATLAB C++ Math Library consists of mathematical functions only. It does not contain any Handle Graphics® or Simulink® functions. Nor does it contain those functions that require the MATLAB interpreter, most notably `eval()` and `input()`. In addition, multidimensional arrays, cell arrays, structures, and objects are not currently supported by the library. Finally, the MATLAB C++ Math Library cannot create or manipulate sparse matrices.

NOTE: Version 1.2 of the MATLAB C++ Math Library is a compatibility release that brings the MATLAB C++ Math Library into compliance with MATLAB 5. Although the MATLAB C++ Math Library is compatible with MATLAB 5, it does not support many of its new features.

Audience

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++. For more information on MATLAB or C++, see one of the references suggested below.

Suggested References

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

Getting Started with MATLAB, The MathWorks

Using MATLAB, The MathWorks

Online *MATLAB Function Reference*, The MathWorks

Overview

This section sketches the shape of the MATLAB C++ Math Library. By no means does it cover all the details. For a comparison between MATLAB and the MATLAB C++ Math Library, see Chapter 2. For complete information about the library, refer to Chapter 4.

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 C++ Math Library class `mwArray` with the 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`.

How This Book Is Organized

This book serves as both a tutorial and a reference. Chapters 1 through 3 are a tutorial. Chapters 4 through 6 and the appendices are reference material.

- **Chapter 1: Getting Ready**: The first chapter provides a brief overview of the MATLAB C++ Math Library and the documentation set, instructions for installing the library, and information on building your own applications.
- **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: Writing Programs**: This chapter presents a series of example programs that demonstrate the basic tasks supported by the library. The final example solves a real-world problem: edge-detection in Microsoft Windows Bitmap files.
- **Chapter 4: Using the Library**: The material in this chapter describes in detail the capabilities of the MATLAB C++ Math Library. Each section addresses a specific topic and is relatively self-contained. You do not need to read every section to use the library effectively. The chapter is primarily intended as reference material.
- **Chapter 5: Library Routines**: This chapter groups the more than 350 library functions into functional categories and provides a short description of each function. The public methods of the `mwArray` class are discussed in detail.
- **Chapter 6: Directory Organization**: Installing the MATLAB C++ Math Library creates several new directories on your computer. This chapter provides a road map to the directories and their contents for IBM PC compatible machines running Microsoft Windows and UNIX workstations.
- **Appendix A: Tips, Tricks and Techniques**: This appendix consists of a series of problems and their solutions. You may not use all of the

demonstrated techniques frequently, but they are sometimes essential to your work with the MATLAB C++ Math Library.

- **Appendix B: The mxArray Structure**: This appendix briefly describes the `mxArray` data structure encapsulated by the `mwArray` class.
- **Appendix C: Error Messages**: This appendix provides a reference to the error messages issued by the library.

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 six sections give you a solid understanding of programming with the MATLAB C++ Math Library.

- The earlier section "Overview" provides an essential overview of the structure of the MATLAB C++ Math Library.
- "Example 4: Handling Errors" in Chapter 3 demonstrates most of the features of the library: creating matrices, writing and calling your own functions, printing matrices, and handling errors.
- "Building C++ Applications" in Chapter 1 explains how to build and run the example programs with the `mbuild` script. This is highly recommended reading.
- "Translating from MATLAB to C++" in Chapter 4 details the differences between MATLAB and C++, and provides information about the MATLAB C++ Math Library.
- "Creating Arrays" in Chapter 4 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 and Subscripts" 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.

Documentation Set

The complete documentation set for the MATLAB C++ Math Library consists of printed and online publications. The online reference documents the C++ Math Library functions themselves.

Primary Sources of Information

- This book, the *MATLAB C++ Math Library User's Guide*
- The online *MATLAB C++ Math Library Reference*
- An online PDF version of the *MATLAB C++ Math Library User's Guide*
- An online PDF version of the *MATLAB C++ Math Library Reference*

Using the Online References

To look up the syntax and behavior for each of the C++ Math Library functions, refer to the online *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 MATLAB user:

1. Type `helpdesk` at the MATLAB Prompt.
2. From the MATLAB Help Desk, select *C++ Math Library Reference* from the **Other Products** section.

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

Additional Sources

- *MATLAB C Math Library User's Guide*
- Online *MATLAB C Math Library Reference*
- *MATLAB Application Program Interface Guide*
- Online *MATLAB Application Program Interface Reference*
- Online *MATLAB Function Reference*
- *Installation Guide for UNIX*
- *Installation Guide for PC and Macintosh*
- Release notes for the MATLAB C++ Math Library

Installing the C++ Math Library

The MATLAB C++ Math Library is available on UNIX workstations and IBM PC compatible computers running Microsoft Windows (Windows 95 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 C++ Math Library. Follow the instructions in the MATLAB *Installation Guide* for your specific platform:

- *Installation Guide for UNIX*
- *Installation Guide for PC and Macintosh*

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

Before you can install the C++ Math Library, you will require an appropriate `FEATURE` line in your License File (UNIX or networked PC users) or an appropriate Personal License Password (non-networked PC users). If you do not yet have the required `FEATURE` line or Personal License Password, contact The MathWorks immediately:

- Via e-mail at `service@mathworks.com`
- Via telephone at 508-647-7000, ask for Customer Service
- Via 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:

- *Installation Guide for UNIX*
- *Installation Guide for PC and Macintosh*

Before you begin installing the C++ Math library, you must obtain from The MathWorks a valid License File (UNIX or networked PC users) or Personal License Password (non-networked PC users). These are usually supplied by fax or e-mail. If you have not already received a License File or Personal License Password, contact The MathWorks by any of these methods:

- Via e-mail at `service@mathworks.com`
- Via telephone at 508-647-7000; ask for Customer Service
- Via 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.

Workstation Installation Details

To verify that the MATLAB C++ Math Library has been installed correctly, use the `mbuild` script, which is documented in "Building on UNIX" on page 1-17, 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`.

PC Installation Details

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 on Microsoft Windows" on page 1-22, to verify that you can build one of the example applications. Be sure to use `mbuild` before calling Technical Support.

You can spot check that the installation worked by examining the file and directory structure:

File	Directory
`matlab.hpp`	`<matlab>\extern\include\cpp`

`libmatpb50.lib`
`libmatpb52.lib`
`libmatpm.lib`
`libmatpw106.lib`
`libmatpw11.lib`
	`<matlab>\extern\lib`

`libmmfile.dll`
`libmatlb.dll`
`libmcc.dll`
	`<matlab>\bin`

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.

Things to Be Aware of 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.
ActiveX	Not required.

Description	Comment
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 explains how to build stand-alone C++ applications on UNIX and Microsoft Windows systems.

The section begins with a summary of the steps involved in building C++ applications with the `mbuild` script and then describes platform-specific issues for each supported platform. `mbuild` helps automate the build process.

You can use the `mbuild` script to build the examples presented in Chapter 3 and to build your own C++ applications. You'll find the source for the examples in the `<matlab>/extern/examples/cppmath` subdirectory; `<matlab>` represents the top-level directory where MATLAB is installed on your system.

Overview

On both UNIX and Microsoft Windows systems, you must follow three steps to build C++ applications with `mbuild`:

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

Once you have properly configured `mbuild`, you simply repeat step 3 to build your applications. You only need to go back to steps 1 and 2 if you change compilers, for example, from Watcom to MSVC, or upgrade your current compiler.

Figure 1-1 shows this the configuration and verification steps on all platforms. The sections following the flowchart provide more specific details for the individual platforms.

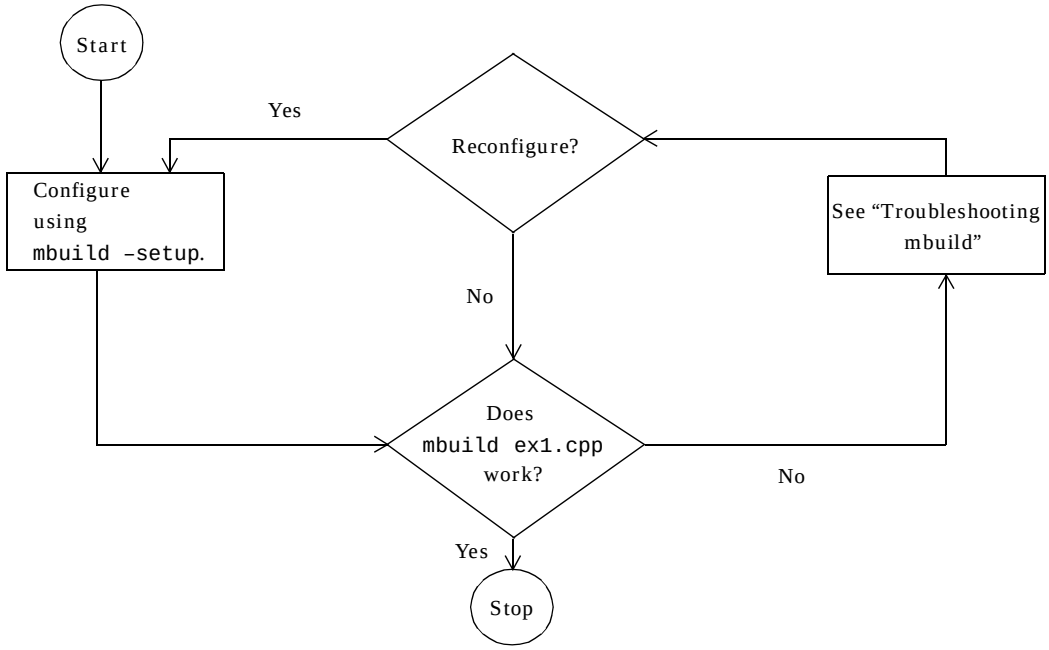


Figure 1-1: Sequence for Creating C++ Applications

Packaging Stand-Alone Applications

To distribute a stand-alone application, you must include the application's executable as well as the shared libraries with which the application was linked against. The necessary shared libraries vary by platform and are listed within the individual UNIX and Windows sections that follow.

Getting Started

In order 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 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

`mbuild` stores your compiler and linker settings in an "options file." Before you can use `mbuild` to create an application, you must first configure it for your system. The configuration process is slightly different for each type of system.

Building on UNIX

This section explains how to compile and link C++ source code into a stand-alone UNIX application.

Configuring mbuild

To configure `mbuild`, at the UNIX prompt type:

```
mbuild –setup
```

The `setup` switch creates a user-specific options file for your compiler.

NOTE: Only the GNU C++ compiler is supported on Sun4.

Executing `mbuild –setup` presents a list of options files.

```
mbuild –setup

The options files available for mbuild are:

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

Enter the number of the options file to use as your default
options file:
```

To select the proper options file for creating a stand-alone C++ application, enter `1` and press **Return**. If an options file doesn't exist in your MATLAB directory, the system displays a message stating that the `mbuildopts.sh` file is being copied to your MATLAB directory. If an options file already exists in your MATLAB directory, the system prompts you to overwrite it.

Changing Compilers. If you want to switch between C and C++, use the `mbuild -setup` command and make the desired changes.

Verifying mbuild

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.

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`, `lnx86`, `sgi`, `sgi64`, `sol2`, or `sun4`)

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.

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:
```

The mbuild Script

The `mbuild` script supports various switches that allow you to customize the building and linking of your code. All users must execute `mbuild -setup` at least once. During subsequent `mbuild`s, the other switches are optional. The `mbuild` syntax and options are:

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

Table 1-1: 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 <file>	Use <file> as the options file; <file> is a full pathname if it is not in current directory. (Not necessary if you use the -setup option.)
-F <file>	Use <file> as the options file. (Not necessary if you use the -setup option.) <file> is searched for in the following manner: The file that occurs first in this list is used: • ./<filename> • $HOME/matlab/<filename> • $TMW_ROOT/bin/<filename>
-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.
-l<file>	Link against library lib<file>.
-L<pathname>	Include <pathname> in the list of directories to search for libraries.
<name>=<def>	Override options file setting for variable <name>.

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

Option	Description
-n	No execute flag. This option causes the commands used to compile and link the target to display without executing them.
-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 switch should be the only argument passed.
-U<name>	Undefine C++ preprocessor macro <name>.
-v	Verbose; print all compiler and linker settings.

Customizing mbuild

If you need to change the switches that `mbuild` passes to your compiler or linker, use the verbose switch, `-v`, as in:

```
mbuild -v filename.cpp [filename1.cpp filename2.cpp …]
```

to generate a list of all the current compiler settings. If you need to change the switches, use an editor to make changes to the options file, `mbuildopts.sh`, which is in your local MATLAB directory (typically `~/matlab`). You can also embed the settings obtained from the verbose switch into an Integrated Development Environment (IDE) or makefile.

`mbuild -setup` copies a master options file to your local MATLAB directory and then edits the local file. If you want to make your edits persist through repeated uses of `mbuild -setup`, you must edit the master file itself called `<matlab>/bin/mbcxxopts.sh`.

NOTE: Any changes that you make to the local options file will be overwritten the next time you execute `mbuild -setup`.

Distributing Stand-Alone UNIX Applications

To distribute a stand-alone application, you must include the application's executable as well as the shared libraries with which the application was linked against. This package of files includes:

- Application (executable)
- `libmatpp.`*`ext`*
- `libmmfile.`*`ext`*
- `libmatlb.`*`ext`*
- `libmcc.`*`ext`*
- `libmat.`*`ext`*
- `libmx.`*`ext`*
- `libut.`*`ext`*

where *`.ext`* is

`.a` on IBM RS/6000 and Sun4; `.so` on Solaris, Alpha, Linux, and SGI; and `.sl` on HP 700.

For example, to distribute the `ex1` example for Solaris, you need to include `ex1`, `libmmfile.so`, `libmatlb.so`, `libmcc.so`, `libmat.so`, `libmx.so`, `libut.so`, and `libmatpp.so`. The path variable must reference the location of the shared libraries.

Building on Microsoft Windows

This section explains how to compile and link C++ code into stand-alone Windows applications.

Configuring mbuild

To configure `mbuild`, at the DOS command prompt type

```
mbuild –setup
```

The `setup` switch creates an options file for your C++ compiler.

You *must* run `mbuild -setup` before you create your first stand-alone application using `mbuild`; otherwise, when you try to create an application, you will get the message:

```
Sorry! No options file was found for mbuild. The mbuild script
must be able to find an options file to define compiler flags and
other settings.  The default options file is
$script_directory\\$OPTFILE_NAME.

To fix this problem, run the following:

mbuild -setup

This will configure the location of your compiler.
```

Refer to "Troubleshooting mbuild" on page 1-29 if you have problems running `mbuild -setup`.

Executing `mbuild -setup` presents you with a list of questions. You will be asked to specify which library to link against and which compiler to use.

This example shows how to select the Microsoft Visual C/C++ compiler:

```
mbuild -setup
Welcome to the utility for setting up compilers
for building math library applications files.

Choose your default Math Library:
[1] MATLAB C Math Library
[2] MATLAB C++ Math Library

Math Library: 2

Choose your C/C++ compiler:
[1] Borland C/C++        (version 5.0 or version 5.2)
[2] Microsoft Visual C++ (version 4.2 or version 5.0)
[3] Watcom C/C++         (version 10.6 or version 11)

[0] None

compiler: 2
```

If we support more than one version of the compiler, you are asked for a specific version. For example,

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

version: 2
```

Next, you are asked to enter the root directory of your ANSI compiler installation:

```
Please enter the location of your C/C++ compiler: [c:\devstudio]
```

Finally, you must verify that the information is correct:

```
Please verify your choices:

Compiler: Microsoft Visual C++ 5.0
Location: c:\devstudio
Library: C++ math library

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

Default options file is being updated...
```

If you respond to the verification question with `n` (no), you get a message stating that no compiler was set during the process. Simply run `mbuild -setup` once again and enter the responses for your system.

Changing Compilers. If you want to change your C++ (system) compiler, make other changes to its options file (e.g., change its root directory), or switch between C and C++, use the `mbuild -setup` command and make the desired changes.

NOTE: An options file is specific to C or C++. You cannot use a C options file with C++ or a C++ options file with C.

Verifying mbuild

There is C++ source code for example `ex1.cpp` 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.

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:
```

The mbuild Script

The `mbuild` script supports various switches that allow you to customize the building and linking of your code. All users must execute `mbuild –setup` at least once. During subsequent `mbuild`s, the other switches are optional. The `mbuild` syntax and options are:

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

Table 1-2: mbuild Options on Microsoft Windows

Option	Description
-c	Compile only; do not link.
-D<name>	Define C++ preprocessor macro <name>.
-f <file>	Use <file> as the options file; <file> is a full path name if it is not in current directory. (Not necessary if you use the -setup option.)
-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.
-output <name>	Create an executable named <name>. (An appropriate executable extension is automatically appended.)
-O	Build an optimized executable.
-setup	Setup location of the installed compiler. This switch should be the only argument passed.
-U<name>	Undefine C++ preprocessor macro <name>.
-v	Verbose; print all compiler and linker settings.

Customizing mbuild

If you need to change the switches that mbuild passes to your compiler or linker, use the verbose switch, -v, as in:

```
mbuild -v filename.cpp [filename1.cpp filename2.cpp …]
```

to generate a list of all the current compiler settings. If you need to change the switches, use an editor to make changes to the options file that corresponds to your compiler. The local options file is called `compopts.bat`. You can also embed the settings obtained from the verbose switch into an Integrated Development Environment (IDE) or makefile. We provide sample project files to help you use your IDE. This is the extent of support provided in this area.

`mbuild -setup` copies a master options file to your local MATLAB directory and then edits the local file. If you want to make your edits persist through repeated uses of `mbuild -setup`, you must edit the master file itself.

Compiler	Master Options File
Borland C++, Version 5.0	`bcccompp.bat`
Borland C++, Version 5.2	`bcc52compp.bat`
Microsoft Visual C/C++, Version 4.2	`msvccompp.bat`
Microsoft Visual C/C++, Version 5.0	`msvc50compp.bat`
Watcom C/C++, Version 10.6	`watccompp.bat`
Watcom C/C++, Version 11	`wat11ccompp.bat`

NOTE: Any changes that you make to the local options file will be overwritten the next time you execute `mbuild –setup`.

Shared Libraries (DLLs)

All the libraries for the MATLAB C++ Math Library are in the directory

```
MATLAB\bin
```

The relevant libraries for building stand-alone applications are WIN32 Dynamic Link Libraries (DLLs). Before running a stand-alone application, you must ensure that the directory containing the DLLs is on your path.

The .def files for the Microsoft and Borland compilers are in the MATLAB\extern\include directory; mbuild dynamically generates import libraries from the .def files.

NOTE: If you want to use an IDE to create your applications, you should look at the project template files and makefiles included in the following compiler-specific directory that corresponds to your compiler.

```
<matlab>\extern\examples\cppmath\borland
<matlab>\extern\examples\cppmath\watcom
<matlab>\extern\examples\cppmath\msvc
```

Distributing Stand-Alone Microsoft Windows Applications

To distribute a stand-alone application, you must include the application's executable as well as the shared libraries against which the application was linked. This package of files includes:

- Application (executable)
- libmmfile.dll
- libmatlb.dll
- libmcc.dll
- libmat.dll
- libmx.dll
- libut.dll

NOTE: The C++ Math Library is static, not shared, so you don't have to distribute it.

In addition, if you created your application with Microsoft Visual C++, you must distribute two MSVC runtime libraries:

- msvcrt.dll
- msvcirt.dll

For example, to distribute the Microsoft Visual C++ version of the `ex1` example, you need to include `ex1.exe`, `libmmfile.dll`, `libmatlb.dll`, `libmcc.dll`, `libmat.dll`, `libmx.dll`, `libut.dll`, `msvcrt.dll`, and `msvcirt.dll`.

The DLLs must be on the system path. You must either install them in a directory that is already on the path or modify the PATH variable to include the new directory.

Troubleshooting mbuild

This section identifies some of the more common problems that might 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, the process will terminate and you will not be able to use `mbuild` to create your applications.

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 Microsoft Windows, if you get errors such as `unrecognized command` 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.

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 508 647-7200.

Building on Your Own

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 Chapter 6 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 Compiler Library (`libmcc`)
4. MATLAB Built-In Library (`libmatlb`)
5. MATLAB MAT-file Library (`libmat`)
6. MATLAB Application Program Interface Library (`libmx`)
7. Standard C Math Library (`libm`)

Specifying the libraries in the wrong order typically causes linker errors. Note that if you are using the Microsoft Visual C/C++ compiler, you must manually build the import libraries from the `.def` files. If you are using the Borland Compiler, you can link directly against the `.def` files. If you are using Watcom, you must build them from the DLLs.

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.

Fundamentals

This chapter 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 chapter begins with a high-level overview of MATLAB. It introduces MATLAB's basic data type, the array, and describes how MATLAB functions and operators perform matrix computation. The next two sections describe the differences between C++ and MATLAB from two perspectives: a C++ programmer's and a MATLAB user's. We assume that you are one or the other. The fourth section describes the basic operation of the MATLAB C++ Math Library and the data types it defines. The chapter concludes with a general description of MATLAB stand-alone programs and a short index to further information.

The sections in this chapter include:

- MATLAB Basics
- MATLAB for C++ Programmers
- C++ for MATLAB Users
- MATLAB C++ Math Library Basics
- Stand-Alone Programs
- Learning More

This sounds like a lot of detail but it really isn't. 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; Chapter 3 for examples; and Chapter 4 for the details this chapter omits.

Some of the material in this chapter will undoubtedly serve as review for many readers. Feel free to skip ahead.

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 Chapter 4.

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 currently only supports the types `double` and `char` and two-dimensional arrays.

A MATLAB matrix is a two-dimensional array of double-precision floating-point numbers. 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 matrix has some basic attributes: the number of rows in the matrix, the number of columns in the matrix, and the array of double-precision floating-point numbers that contains the data in the matrix. The data in a matrix can be either real or complex numbers. If a matrix 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 a matrix 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 “Creating Arrays” in Chapter 4 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. For example, `A * B` is the linear algebraic product (matrix multiplication) of `A` and `B`, while `A / B` is equivalent to `(A' / B')'`.

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 "Calling Conventions" in Chapter 4 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 5: Using load() and save()" in Chapter 3 and "Example 6: Using File I/O Functions" in Chapter 3 and 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" in Chapter 4 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, a group of elements or an entire row or column of 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:

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

writes the value `13` into the entry at row four and column seven of array `A`. This is the only way to modify the contents of an array.

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 "Translating from MATLAB to C++" in Chapter 4 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.
- 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 "Translating from MATLAB to C++" in Chapter 4 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; Chapter 4 provides much more depth. Before you read this section, make sure you understand all of the concepts in "MATLAB Basics" in Chapter 2.

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.

NOTE: The MATLAB C++ Math Library supports at most two dimensions for an array.

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 a number of rows, a number of columns, and either one or two arrays of data. Each of these

arrays has `row * columns` elements. 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. "mwArray Class Interface" in Chapter 5 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 "Indexing and Subscripts" 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 (see "Operators" in Chapter 2). 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 5 lists the available operators and their function call equivalents.

Functions

The MATLAB C++ Math Library contains over 350 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 "Calling Conventions" in Chapter 4 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.

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 6: Using File I/O Functions" in Chapter 3 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 "Example 5: Using load() and save()" in Chapter 3 or "Using load( ) and save( )" in Chapter 4 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. “Array Input and Output” in Chapter 4 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 “Exception Handling and Error Messages” in Chapter 4 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 4 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 "Distributing Stand-Alone UNIX Applications" on page 1-22 and "Distributing Stand-Alone Microsoft Windows Applications" on page 1-28.

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 with no graphical output. 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.

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.

- **Calling Functions**

  "Calling Conventions" in Chapter 4

  "Example 2: Writing Simple Functions" in Chapter 3

  "Example 3: Calling Library Routines" in Chapter 3

- **Errors**

  "Example 4: Handling Errors" in Chapter 3

  "Exception Handling and Error Messages" in Chapter 4

- **Indexing or Subscripting**

  "Indexing and Subscripts" in Chapter 4

  "Duplicating a Row or Column" in Appendix A

- **Input and Output**

  "Example 5: Using load() and save()" in Chapter 3

  "Example 6: Using File I/O Functions" in Chapter 3

  "Array Input and Output" in Chapter 4

- **Arrays and Matrices**

  "Example 1: Creating Arrays and Array I/O" in Chapter 3

  "Creating Arrays" in Chapter 4

  "Data Conversions" in Chapter 4

  "The mxArray Structure" in Appendix B

- **Operators**

  "Operators" in Chapter 5

  "Mathematical Operators" in Chapter 4

- **Syntax**

  "Translating from MATLAB to C++" in Chapter 4

  "Example 7: Rewriting roots.m in C++" in Chapter 3

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) 647-7200 (voice)
  (508) 647-7201 (FAX)
- US Mail:
  The MathWorks, Inc.
  24 Prime Park Way
  Natick, MA 01760-1500

3

Writing Programs

The best way to learn how to use the MATLAB C++ Math Library is to see the library in use. This chapter contains nine examples, presented in order of increasing complexity. Each annotated example illustrates a particular feature of the library. The subjects of the examples are:

- Creating arrays and array I/O
- Writing simple functions
- Calling library routines
- Handling errors
- Using `load()` and `save()`
- Using file I/O functions
- Rewriting `roots.m` in C++
- Passing functions as arguments
- Using the MATLAB Compiler

Each of these examples presents a complete, working program. The numbers to the left of code statements reference annotations presented in a "Notes" section that immediately follows each example. An "Output" section that shows the output produced by the example is presented next, followed by a short summary titled "What the Example Demonstrates." Scanning each "What the Example Demonstrates" section provides an excellent, quick overview of the MATLAB C++ Math Library.

You can find the code for each example in the `<matlab>/extern/examples/cppmath` directory where `<matlab>` represents the top-level directory of your installation. See "Building C++ Applications" in Chapter 1 for information on building the examples.

NOTE: Think of the `mwArray` objects in the examples as actual arrays rather than pointers to arrays. Array arguments are conceptually passed by value. The implementation of the `mwArray` class is such that pass-by-value is not expensive, even for large arrays. See "Calling Conventions" in Chapter 4 for more details on this subject.

Example 1: Creating Arrays and Array I/O

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.

```
// 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;
(6) cout << mat1 << endl;

    return (EXIT_SUCCESS);
}
```

Notes

The numbers in this list refer to the numbered areas of code above.

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 and IBM-PC systems read from the terminal by default; you can redirect them to read from an input file. See "Array Input and Output" in Chapter 4 to learn about the I/O format for the library and how it differs from MATLAB's.

6 Print the newly read matrix.

Output

During Step 4, the program prints:

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

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

In Step 5, the program prints "`Please enter a matrix:`" and waits for you to type something. 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. Because each matrix is delimited by `[` and `]`, input files or streams can contain more than one matrix.

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.

See "Array Input and Output" in Chapter 4 for a more complete description of the array input and output format.

What the Example Demonstrates

- All programs written using the MATLAB C++ Math Library must include the header file `matlab.hpp`.
- You can create simple arrays from static data arranged in column-major order. To arrange array data in column-major order, read down the columns of the array from the leftmost column to the rightmost column.
- Using C++ iostreams, you can write a simple array to a file and read it back again.
- 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.

Example 2: Writing Simple Functions

This example demonstrates how to write a simple function that takes two matrix arguments and returns a matrix value.

```
// ex2.cpp

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

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

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

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

(6)     mat2 = average(mat0, mat1);

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

        return(EXIT_SUCCESS);
    }
```

Notes

The numbers in this list refer to the numbered sections of code above.

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 1: Creating Arrays and Array I/O" on page 3-3, 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
]
```

What the Example Demonstrates

- 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.

Example 3: Calling Library Routines

There are subtleties of the library interface that the previous example did not cover. 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.

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()`.

```
// ex3.cpp

#include <stdlib.h>
#include "matlab.hpp"

static double data[] = { 1, 3, 5, 7, 2, 4, 6, 8 };

int main(void)
{
    // Create the input matrix.
    mwArray X(4, 2, data);
    mwArray U, S, V;

    // Compute the singular value decomposition of the matrix.
    cout << "One input, one output: " << endl ;
    cout << "S = " << svd(X) << endl;

    // Pass in optional output arguments.
    U = svd(&S, &V, X);

    cout << "One input, three outputs: " << endl;
    cout << "U = " << U << "S = " << S << "V = " << V << endl;

    // Pass in optional input argument.
    U = svd(&S, &V, X, 0.0);

    cout << "Two inputs, three outputs: " << endl;
    cout << "U = " << U << "S = " << S << "V = " << V;

    return(EXIT_SUCCESS);
}
```

Notes

The numbers in this list refer to the numbered areas of code above.

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 1: Creating Arrays and Array I/O" on page 3-3 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 “Array Input and Output” in Chapter 4 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
  ]
```

What the Example Demonstrates

- 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.

Example 4: Handling Errors

This example demonstrates the MATLAB C++ Math Library's error handling facilities. The program uses a negative number as an array index, which causes an error. The MATLAB C++ Math Library then raises a C++ exception.

Exceptions are a relatively new addition to many C++ compilers. Be sure your compiler supports them robustly. To learn more about C++ exception handling, read the short introduction in "Exception Handling and Error Messages" in Chapter 4, or consult your C++ reference.

```
// ex4.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.
    return in(-5) * 17;
}

int main(void)
{
    // Handle exceptions for all code in the try-block.
    try {
        mwArray mat0(2, 3, data);
        mwArray mat1;

        mat1 = compute(mat0);
        cout << mat1 << endl;
    }
    // Catch and print any exceptions that occur.
    catch (mwException &ex) {
        cout << ex << endl;
        return(EXIT_FAILURE);
    }
    return(EXIT_SUCCESS);
}
```

Notes

The numbers in the list below correspond to the numbered sections of code above.

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 doubles. Its elements are specified in column-major order. Refer to "Example 1: Creating Arrays and Array I/O" on page 3-3 for more detail on column-major element order and why you should use C++ one-dimensional arrays to initialize MATLAB C++ Math Library array data.

3 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()`.

See "Example 2: Writing Simple Functions" on page 3-7 for more information about writing functions using the MATLAB C++ Math Library. Refer to "Indexing and Subscripts" in Chapter 4 for complete details on the MATLAB C++ Math Library's indexing mechanism.

4 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.

5 Declare the input matrix, `mat0`, and the matrix that stores the result, `mat1`.

6 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.

7 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.

This example is straightforward. It 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.

What the Example Demonstrates

- The MATLAB C++ Math Library uses C++ exceptions to report errors.
- Your programs *must* catch exceptions thrown by the MATLAB C++ Math Library. See "Exception Handling and Error Messages" in Chapter 4 for a complete list of exceptions.
- Uncaught exceptions cause abnormal program termination.
- 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.

Example 5: Using load() and save()

This example demonstrates how to use the functions `load()` and `save()` to write your data to a disk file and read it back again. `load()` and `save()` operate on MAT-files, which use a special binary file format that ensures efficient storage and cross-platform portability. MATLAB can read and write MAT-files, too, so 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.

The MATLAB C++ Math Library functions `save()` and `load()` implement the MATLAB `load` and `save` functions. Note, however, that not all the variations of the MATLAB `load` and `save` syntax are implemented for the MATLAB C++ Math Library. See the section “Array Input and Output” in Chapter 4 for further information on these two functions.

```
// ex5.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);
}
```

Notes

The numbers in the list below correspond to the numbered sections of code above.

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.
```

What the Example Demonstrates

- `load()` and `save()` let you share array data with other MATLAB C and C++ Math Library applications and with MATLAB applications.
- 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.

Example 6: Using File I/O Functions

MATLAB C++ Math Library file I/O functions, such as `fprintf()`, are similar to the ANSI standard C functions of the same name, with several significant restrictions and extensions. This example demonstrates how to use `fopen()`, `fclose()`, `fprintf()`, `fgetl()`, and `fscanf()`.

```
// 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)     // Write string data to a file. Then read it.
(4)     fid = fopen(file, mode);
(5)     fprintf(fid, "%s\n", a, b, c, d);
        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);
```

```
        // Now try numeric data.
        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);
    }
```

Notes

The numbers in the list below correspond to the numbered sections of code above.

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.

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
  ]
```

Because the MATLAB C++ Math Library file I/O functions have the same name as their counterparts, the C++ file I/O functions, 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.

This short example does not demonstrate all the features of `fprintf()`. See the online *MATLAB C++ Math Library Reference* for complete details.

What the Example Demonstrates

- The MATLAB C++ Math Library supports C and C++-like file I/O with the functions `fprintf()`, `fgetl()`, `fgets()`, `fopen()`, `fclose()`, `fscanf()` and `fprintf()`.
- The vectorized versions of `fprintf()` and `fscanf()` in the MATLAB C++ Math Library take matrices as arguments, and repeatedly recycle their format strings through the matrices to produce the output or read the input.
- The first argument to `fprintf()` and `fscanf()` *must* be an `mwArray` object.
- You can construct an `mwArray` object from a string.
- The versions of `fprintf()` and `sprintf()` in the MATLAB C++ Math Library take up to 32 arguments.
- `fgetl()` and `fgets()` work correctly on ASCII files only.
- By default, `fopen()` opens files in read-only mode.

Example 7: Rewriting roots.m in C++

MATLAB's `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 MATLAB 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 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.

```
// Call roots() using the example in the MATLAB Users Guide.
int main(void)
{
    // Static array of doubles used to initialize the matrices.
(1) static double input[] = { 1, -6, -72, -27 };

    // Declare three matrices, one with initial values.
(2) mwArray x(1, 4, input), result, verify;

    // Call our version of roots().
(3) result = example_roots(x);

    // Call the MATLAB C Math Library roots.
(4) verify = roots(x);

    // Print the input and output matrices from example_roots().
    cout << "x = " << endl << x << endl;
(5) cout << "example_roots(x) = " << endl << result << endl;

    // Check to see if the answer is equal to the real roots().
(6) if (tobool(result == verify))
        cout << "Success!" << endl;

    return(EXIT_SUCCESS);
}
```

Notes

The numbers in the list below correspond to the numbered sections in the main program above. The main program is straightforward, so the explanations below are brief. If you have difficulty understanding this section of the example, refer to the previous examples, in particular "Example 2: Writing Simple Functions" on page 3-7.

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>
#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).
mwArray example_roots(mwArray c)
{
    mwArray n, inz, nnz, r, a;
    mwIndex icolon;

    // Make sure # of dimensions is not greater than 1
    n = size(c);
    if (all(n > 1.0))
        error("Must be a vector");

    n = max(n);
    c = transpose(c(icolon)); // Make sure it's a row vector.
    inz = find(abs(c));       // Find all nonzero elements.
    nnz = max(size(inz));     // Count nonzero elements.

    // Test all elements against zero.
    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.
    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.
    if (n > 1.0)
        a(1,icolon) = -c(ramp(2, n)) / c(1);
    r = vertcat(r, eig(a));

    return r;
}
```

Notes

The numbers in this list correspond to the numbered sections of code in the `example_roots()` function.

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 "Indexing and Subscripts" 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!
```

The MATLAB roots() Function

The C++ `example_roots()` function is a translation of the MATLAB `roots()` function. For purposes of comparison, `roots.m`, the M-file that contains the `roots()` function, 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)];
```

What the Example Demonstrates

- 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.
- Calling the `error()` function throws an `mwRuntimeException` exception. `mwRuntimeException` is a subclass of `mwException`.
- `vertcat()` vertically concatenates two or more (up to ten) 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.

Example 8: Passing Functions As Arguments

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.

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 `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.

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 8, 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);
}
```

Notes

The numbers in the list below correspond to the numbered sections of code above.

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()`.

```
// ex8.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);
    }
```

Notes

The numbers in the list below correspond to the numbered sections of code above.

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;
```

Notes

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
(4)  int one_input_one_output(mlfFuncp pFunc, int nlhs,
                              mxArray **lhs, int nrhs,
                              mxArray **rhs)
     {
         mwArray Out;

(5)      if ( nlhs > 1 || nrhs > 1 )
         {
             return(0);
         }

(6)      mwArray tmp = mwArray( rhs[0], 0 );

(7)      Out = (*((PFCN_1_1)pFunc))( nrhs > 0 ? tmp
                                              : mwArray::DIN);

         if (nlhs > 0)
         {
(8)          lhs[0] = Out.FreezeData();
         }

(9)      return(1);
     }
     }
```

Notes

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 4 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
]
```

What the Example Demonstrates

- `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.

Example 9: Using the MATLAB Compiler

The MATLAB Compiler turns M-files into C or C++ code. This example demonstrates how to use the MATLAB Compiler to produce a stand-alone C++ executable from a collection of M-files. You can also use these techniques to incorporate MATLAB functions into a pre-existing C++ program. Note that this example requires that you own the MATLAB Compiler.

Most of the functions in the MATLAB Toolboxes are not present in the C++ Math Library; however, the MATLAB Compiler makes them available to C++ programs. For example, the Image Processing Toolbox provides an edge-detection routine, `edge()`. This example uses `edge()` and other routines from the Image Processing Toolbox to perform edge detection in Microsoft Windows Bitmap files.

NOTE: Version 4 of the MATLAB Image Processing Toolbox, *not* version 5, is the source for the image processing routines used in this example. The code included here is MATLAB 4 code.

The program consists of five steps:

1. Read a Microsoft Windows Bitmap file.
2. Convert the bitmap image into a grayscale image.
3. Perform edge-detection on the grayscale image.
4. Convert the resulting black-and-white Boolean mask into an image.
5. Write the image out to a new Microsoft Windows Bitmap file.

These steps require direct calls to five Image Processing Toolbox routines: `bmpread()`, `ind2gray()`, `edge()`, `gray2ind()`, and `bmpwrite()`. As a group, these routines call five other Image Processing routines: `bestblk()`, `gray()`, `grayslice()`, `rgb2ntsc()`, and `fspecial()`. These last five routines don't call any other M-files, though they do call routines built into the C++ Math Library. Therefore, 10 M-files need to be compiled to build this example. However, since we need to compile these M-files into stand-alone code, they need to be modified slightly. These modifications are necessary because the M-files used in this

example make calls to Handle Graphics® routines, none of which exist in the MATLAB C++ Math Library. The code for the 10 M-files that are compiled is not included in this chapter; however, the M-files are included in the `examples` directory of your MATLAB installation.

If you own the MATLAB Compiler, you can generate `ex9.cpp` with the following command:

```
mcc -phm ex9 gray
```

This command compiles the M-file `ex9.m` and all other M-files called by `ex9`, searching for them on the MATLAB path, and collects the output into a C++ file called `ex9.cpp`. The name of the output file derives from the name of the M-file, `ex9`, listed on the command line. Note that it is not necessary to specify the `.m` filename extension. The `-p` switch instructs the MATLAB Compiler to generate C++ code. The default output language (without the `-p` switch) is C. The `-h` switch instructs the compiler to find all the necessary helper functions automatically. Finally, the `-m` switch tells the compiler to produce a main program; the resulting C++ file can be compiled into a complete stand-alone application.

The MATLAB Compiler can only compile calls to functions for which it either has an internal table entry or that it can find on the MATLAB path. Since the M-files in this example are modified versions of the MATLAB 4.2 Image Processing Toolbox, it is very important that you have the `examples` directory on your MATLABPATH when you compile `ex9.m`. If you do not, it is possible that the MATLAB Compiler will either not find the required M-files, or that it will find files of the same name that will not work with this example.

Type `help mcc` or see the *MATLAB Compiler User's Guide* for more details on the MATLAB Compiler.

`ex9.cpp` is broadly divided into three sections: declarations, compiled helper functions, and the main routine. The first section includes the required header file and provides declarations of constants and functions defined in `ex9.cpp`. As shown below, the MATLAB Compiler declares constants in a two step process.

```
/* static array S5_ (3 x 3) int, line 167 */
static int S5r_[] =
{
        0, 1, 0, 1, 0, 1, 0,
        1, 0,
};
static mwArray S5_ = mwArray( 3, 3, S5r_ );
```

The comment that precedes this constant indicates the constant is a 3-by-3 array of integers (note, however, that the MATLAB C++ Math Library stores integer arrays as arrays of double-precision floating-point numbers) from line 167. The following code appears on line 167 of `edge.m`:

```
b = filter2([0 1 0;1 0 1;0 1 0],bb);
```

This constant represents the first argument in the call to `filter2()`. The first step in declaring the constant is to create a static C++ array that contains the data; as usual, this data is organized in column-major order. The second step is to initialize a static `mwArray` with the data from the static C++ array.

The MATLAB Compiler is very careful to declare all functions before their first use. Here is the declaration of the helper function `edge()`:

```
static mwArray edge(mwArray *,mwArray =mwArray::DIN,
                    mwArray =mwArray::DIN,
                    mwArray =mwArray::DIN,
                    mwArray =mwArray::DIN);
```

There are two interesting aspects of this declaration. Notice first that the Compiler declares `edge()` as a static function. This means that `edge()` can only be called from within this file. If you want to compile a function so that it can be called from more than one file, compile it by itself rather than as a helper function. Also note that this function has five arguments, but that the last four are optional. The MATLAB Compiler uses the special variable `mwArray::DIN` as the default value for optional arguments. Since all optional arguments have a special default value, the MATLAB Compiler can check the values of the input arguments against `mwArray::DIN` and thereby count the number of arguments actually passed to the function. Just as MATLAB uses `nargin` and `nargout` to count the number of input and output variables passed to a function, the MATLAB Compiler uses the variables `_nargin_count` and `_nargout_count`.

The next section of ex9.cpp contains the compiled helper functions. These helper functions are the compiled versions of the M-files called by ex9(). In C++ code generated by the MATLAB Compiler, the only difference between a helper function and any other function is that every helper function is declared static, as shown above. This is the definition of the edge() function, minus most of the body, which is too long to reproduce here:

```
mwArray edge(mwArray *tol_out_, mwArray a, mwArray tol,
             mwArray method, mwArray k)
{
   // Code omitted

   // Begin nargin() counting code.
   int _nargin_count = 0;

   if (a != mwArray::DIN) _nargin_count++;
   else a = mwArray();

   if (tol != mwArray::DIN) _nargin_count++;
   else tol = mwArray();

   if (method != mwArray::DIN) _nargin_count++;
   else method = mwArray();

   if (k != mwArray::DIN) _nargin_count++;
   else k = mwArray();

  // More code omitted
}
```

Notice that there are five arguments, and that the default values for optional arguments do not appear in the definition; C++ requires them in the declaration, so that they are known at the call site. However, the body of the function contains a block of code that counts the number of input arguments with non-default values (i.e., those input arguments whose value is not equal to mwArray::DIN). The MATLAB Compiler detected the use of nargin in edge.m, and automatically generated C++ code to count the number of input arguments. If edge.m had used nargout, the MATLAB Compiler would have generated code to count the number of output arguments as well. Just as mwArray::DIN is the default value for optional input arguments, NULL is the default value for optional output arguments.

The final section of ex9.cpp is the main routine:

```
(1) int main(int argc, char *argv[])
    {

    #ifdef EXCEPTIONS_WORK
(2)    try {
    #endif

(3)    mwArray TempMatrix_[32];
       mwArray infile;
       mwArray outfile;
       mwArray x;
       mwArray map;
       mwArray inten;
       mwArray bw;

(4)    infile = (argc >1) ? mwArray(argv[1]) : mwArray::DIN;
       outfile = (argc >2) ? mwArray(argv[2]) : mwArray::DIN;
       // EX9 Edge Detection on Microsoft Windows bitmap files

       // EX9( INFILE, OUTFILE )
       // Opens and reads the Windows bitmap stored in INFILE and
       // writes the resulting bitmap into OUTFILE.

        // Copyright (c) 1997 by The MathWorks, Inc.
(5)    x = bmpread(&map,0,infile);
(6)    inten = ind2gray(x,map);
(7)    bw = edge(0,inten);
(8)    x = gray2ind(&map,bw);
(9)    bmpwrite(x,map,outfile);

    #ifdef EXCEPTIONS_WORK
(10)   } catch(mwException &ex) { cout << ex << endl; }
    #endif

       return 0;
    }
```

Notes

Each of the numbered steps is explained in more detail below. Steps 5 through 9 correspond to the five basic steps of the image processing program outlined at the beginning of this example.

1. Begin the `main()` function. `ex9` is called with two arguments: the name of the input file and the name of the output file. The `main()` function could just as easily be rewritten as a subroutine of a larger program.

2. Begin the `try`-block. Because the code in the MATLAB C++ Math Library throws exceptions to indicate errors, a `try`-block within the main routine must enclose all calls to the MATLAB C++ Math Library routines (even those routines called indirectly). Establishing a `try`-block permits the program to catch any exceptions that the MATLAB C++ Math Library might throw. See "Exception Handling and Error Messages" on page 4-62 for more details on exception handling.

3. Declare local variables that will store results from the edge detection steps. `x` stores the image data, `map` the colormap, `inten` the grayscale intensity image derived from the original bitmap, and `bw` the black-and-white intensity image returned by `edge()`. `infile` and `outfile` store the names of the input and output bitmap files, and `TempMatrix_` is a variable automatically generated by the MATLAB Compiler.

4. Obtain input and output file names from the command line arguments. Since the first argument, stored in `argv[0]`, is always the name of the program being run, `argv[1]` contains the name of the input file, and `argv[2]` contains the name of the output file. Note that this code sets the file names to a default value if the corresponding argument was not supplied. Later checks will generate errors if either the input file or the output file is omitted.

5. Read in the bitmap. Assume the first command line argument is the name of an input file containing a Microsoft Windows Bitmap. `bmpread()` will fail if this is not the case. Store the image data in `x` and the colormap in `map`.

6. Convert the bitmap to a grayscale intensity image. `ind2gray()` returns a matrix the same size as the input image where all the pixel values range from `0.0` (no intensity, or black) to `1.0` (full intensity, or white).

7 Perform Sobel edge detection on the image. The edge detection routine supports four methods of edge detection: Sobel, Roberts, Prewitt, and Marr-Hildreth. Sobel is the default because it gives consistently good results; it finds edges where the first derivative of the image intensity is maximal or minimal.

8 Convert the grayscale intensity image to a black-and-white indexed image. The image returned by `edge()` contains only 1's (edge) and 0's (background). `gray2ind()` converts this intensity image into an indexed image and computes the associated colormap.

9 Write the image out to a bitmap file. Use the indexed image data and colormap generated by `gray2ind()`. Assume that the second argument on the command line (`argv[2]`) is the name of a file in which to write the output. Destroy any data currently in this file; if the file does not exist, create it.

10 Catch exceptions with a `catch`-block. If an exception occurs anywhere in the program, control resumes here. Print out the message associated with the exception and then exit the program.

Now that you have a better understanding of the program, you can build and run it. See the section, "Building C++ Applications" in Chapter 1 for details on how to build this example. Make sure the file `trees.bmp` is in the directory where you build the executable; copy it from the `examples` directory if necessary.

Run the program by typing:

```
ex9 trees.bmp edges.bmp
```

at your system prompt.

If no errors occur, the program runs without printing any output. If all goes well, the program creates a file called `edges.bmp` in the current directory. This file contains the results of performing Sobel edge detection on the standard MATLAB image called `trees`. Verify that the program worked by viewing the image in `edges.bmp`; almost any graphically-oriented Microsoft program (Paintbrush, MS Paint, etc.) will display a Microsoft Windows Bitmap file. On UNIX systems use a program like *xv* to view the image.

Compiler-Generated Code

The MATLAB Compiler generates C++ code that is clearly written by a program rather than a programmer. This is most evident in the Compiler's use of static variables and its handling of optional input and output arguments.

The MATLAB Compiler turns every string and numeric matrix constant in a MATLAB M-file into two static variables. The first variable is an array of doubles; this is the data corresponding to the MATLAB constant. The second variable is an `mwArray`; it is built using the data in the first variable. The first variable only exists to initialize the second; after the initialization, the program uses the second variable exclusively. The data array is always an array of doubles, regardless of the type of data in the MATLAB constant. String constants become arrays of ASCII numeric values.

The MATLAB Compiler uses C++ default arguments to handle the optional input and output arguments of a MATLAB M-file function. In C++, arguments with default values need not be specified when the function is called. The MATLAB Compiler uses 0 (zero) for optional output arguments and the special array `mwArray::DIN` for optional input arguments. The function declaration specifies which arguments have default values.

In MATLAB, functions with optional arguments use the two special variables `nargin` (number of inputs) and `nargout` (number of outputs) to determine the number of arguments that have been passed. When the MATLAB Compiler compiles a function that uses `nargin` or `nargout`, it generates code to count the number of input and output arguments that don't have the default values. It stores the results in two special variables `_nargin_count` and `_nargout_count`, which correspond to the MATLAB variables `nargin` and `nargout`.

See the *MATLAB Compiler User's Guide* for a complete discussion of the C++ code generated by the Compiler.

What the Example Demonstrates

- The MATLAB Compiler extends the functionality of the MATLAB C++ Math Library by compiling M-files into C++ code.
- The MATLAB Compiler is not shipped with the MATLAB C++ Math Library; the compiler is a separate product available from The MathWorks.
- The MATLAB Compiler and the MATLAB C++ Math Library use different mechanisms to handle optional arguments. The Compiler uses `nargin/`

nargout counting code and default arguments, while the MATLAB C++ Math Library provides multiple overloaded versions of any function that has optional arguments.

- _nargin_count is the variable the MATLAB Compiler uses to accumulate the number of input arguments passed to a function.
- _nargout_count is the variable the MATLAB Compiler uses to accumulate the number of output arguments passed to a function.
- mwArray::DIN is the default input array used to initialize optional input arguments.
- NULL is the default value for output array pointers.
- The MATLAB Compiler cannot compile calls to functions for which it cannot determine an argument list. If the function is a built-in, it must be present in the compiler's internal table. If the function is in an M-file, that M-file must be on the MATLAB path. There are a number of built-in functions that the Compiler cannot handle. See your *MATLAB Compiler User's Guide* for more information on those functions.

4

Using the Library

This chapter describes the technical details of the MATLAB C++ Math Library. The library is designed to make writing numerical programs with C++ as easy as it is with MATLAB M-files. This chapter points out the differences between C++ programming and M-file programming; in this way, MATLAB programmers can use this chapter to learn about C++, and C++ programmers can use it to learn about MATLAB.

You do not need to read this chapter straight through. Read the first section, which explains the salient differences between programming in MATLAB and programming in C++, but after that, read this chapter at your own pace. Treat it as a reference guide, reading only those sections that pertain to the work you're doing.

This chapter explains how to:

- Translate from MATLAB to C++
- Call the C++ Math Library functions
- Create arrays
- Use subscripts to index into an array
- Convert to and from scalars and arrays
- Use the << and >> operators with arrays
- Use the `load()` and `save()` functions
- Write your own print handler for output to a GUI
- Use the mathematical operators
- Handle the exceptions generated by the library
- Provide your own memory management routines
- Improve the performance and efficiency of your application

Translating from MATLAB to C++

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
- Assignment statements, including assignment with an 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. Refer to the release notes for the MATLAB C++ Math Library to find out which MATLAB V5 features the library supports.

Differences Between C++ and MATLAB

Programming in C++ differs from programming in MATLAB in five important areas: syntax, variable declaration, function calling conventions, control structure, and treatment of logical values.

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.

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 "Calling Conventions" in Chapter 3.

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 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 4-1: Renamed Functions in the MATLAB C++ Math Library

MATLAB Name	C++ Math Library Name
`and`	`and_func`
`char`	`char_func`
`clock`	`clock_func`
`double`	`double_func`
`not`	`not_func`
`or`	`or_func`
`pascal`	`pascal_func` (PC only)
`quad`	`quad_func`
`rand`	`rand_func` (`rand()` is also available if you pass one or two arguments)
`union`	`union_func`
`xor`	`xor_func`

Calling Conventions

The MATLAB C++ Math Library includes over 350 functions. This section 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.

You'll find a complete reference for the library functions in the online *MATLAB C++ Math Library Reference* accessible from the Help Desk. That reference lists the arguments and return value for each function, shows you how to call each overloaded version of a function, and lets you access the documentation for the MATLAB version of the function.

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++.

One Return and One or More 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.

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.

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.

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.

Mapping Rules

Two simple 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.

The following table summarizes the mapping between interpreted MATLAB functions and the same functions in the MATLAB C++ Math Library.

Table 4-2: 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

Table 4-2: MATLAB and C++ Function Calling Conventions (Continued)

MATLAB Calling Sequence	C++ Calling Sequence	Input/Output Count
`[ 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

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 5 lists the MATLAB operators and the corresponding MATLAB C++ Math Library functions.

Exceptions to the Calling Conventions

The `load()` and `save()` functions do not follow the standard calling conventions for the library. Not all arguments to the two functions are of type `mwArray` or `mwArray *`. The argument list for each function includes pairs of arguments where the argument representing the name of the variable is a `const char *`. In addition, the standard order for output and input arguments in not followed: `load()` intersperses input and output arguments.

“Example 5: Using load() and save()” in Chapter 3 demonstrates how to call the functions.

Creating Arrays

In MATLAB, there are five ways to create an array. You can:

- Use a creation function, such as `ones()` or `magic()`.
- Specify an array with the `[]` syntax, for example, `A = [1 3 4 6];`.
- Use the colon operator, for example, `1:10`, which creates a vector containing the numbers 1 through 10.
- Assign a scalar to a named variable, for example, `A = 5;`.
- Invoke a function or operation on other arrays.

The MATLAB C++ Math Library supports these five methods and one additional method unique to C++: the C++ constructor. Because C++ does not support the MATLAB-specific operators `:` and `[],` the C++ support for these two creation methods differs from MATLAB's support.

To create an array in C++:

- Use a creation function, such as `ones()` or `magic()`.
- Call `vertcat()` or `horzcat()`, the C++ replacements for MATLAB's `[]` syntax.
- Call `ramp()` or `colon()`, the C++ replacements for MATLAB's colon operator. Use `ramp()` to create sequences of numbers; use `colon()` in indexing expressions.
- Assign a scalar to a named variable, for example, `A = 5;`.
- Invoke a function or operation on other arrays.
- Use a C++ constructor with an optional array of data.

Creation Functions

The MATLAB C++ Math Library includes the MATLAB array construction functions: `ones()`, `eye()`, `rand()`, `magic()`, and `zeros()`. These functions construct some of the more common arrays.

- `ones()` creates a matrix containing all 1's.
- `eye()` creates an identity matrix.
- `rand()` creates a matrix of random numbers between 0 and 1.

- `magic()` creates a matrix square.
- `zeros()` creates a matrix containing all 0's.

Each function, except for `magic()`, accepts either one or two arguments. Given one argument, each function returns a square array; given two arguments, each function returns an array with the indicated number of rows and columns. Because a magic square is always a square array, `magic()` requires only a single argument.

vertcat() and horzcat()

Vertical and horizontal concatenation are useful ways to construct matrices of any size and shape. In MATLAB, the `[]` operator performs both vertical and horizontal concatenation. In C++, you use the functions `vertcat()` and `horzcat()` instead.

For example, the MATLAB statement

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

horizontally concatenates the numbers 1, 2 and 3 into a vector containing one row and three columns:

```
1 2 3
```

The MATLAB statement

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

creates an array with two rows and three columns:

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

Note that a semicolon separates the rows that are vertically concatenated.

The MATLAB `[]` operator works on arrays as well as numbers. For example, if `A = [ 1 2 3 ]` and `B = [ 4 5 6 ]`, then

```
C = [ A ; B ]
```

produces:

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

In the MATLAB C++ Math Library, the mwArray constructors and the functions horzcat() and vertcat() combine to provide the functionality of the MATLAB [] operator. horzcat() concatenates two arrays horizontally, and vertcat() concatenates two arrays vertically. For example, the C++ code:

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

produces a vector of one row and three columns:

```
1 2 3
```

The horzcat() and vertcat() functions expect arguments of type mwArray, and both return an mwArray object. horzcat() and vertcat() each take up to 32 arguments.

You can also nest horzcat() and vertcat() calls to build matrices incrementally. For example, the C++ code

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

produces a matrix of two rows and three columns:

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

Finally, like the MATLAB [] operator, horzcat() and vertcat() also concatenate arrays. If `A = horzcat(1, 2, 3)` and `B = horzcat(4, 5, 6)`, then

```
C = vertcat(A, B);
```

produces the 2-by-3 array:

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

NOTE: Horizontally concatenated arrays must have the same number of rows. Vertically concatenated arrays must have the same number of columns.

For example,

```
C = horzcat(A, B);
```

successfully joins the 2-by-2 array A

```
7 7
8 8
```

to the 2-by-3 array B

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

producing:

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

vertcat(A, B), however, would fail because A has two columns and B has three.

ramp() and colon()

The : (colon) operator is one of the most useful pieces of MATLAB syntax. It creates vectors and acts like a wildcard in indexing expressions. The colon operator also specifies the bounds of MATLAB for-loops.

The MATLAB C++ Math Library emulates the : operator with the ramp() and colon() functions. You use the ramp() function to create vector and the colon() function to index into arrays. Note that the colon() function provided by the library cannot be used to specify the bounds of a C++ for-loop.

The table below demonstrates how to use the two functions. Also refer to "Indexing and Subscripts" on page 4-21 for more information on indexing expressions and to "Generating Sequences" in Chapter 5 and "Indexing" in Chapter 5 for descriptions of the functions.

Table 4-3: Functional Equivalents for the Colon Operator

To...	C++ Example	MATLAB Equivalent
Flatten a matrix into a vector	`B = A(colon());`	`B = A(:);`
Extract the first 15 elements of a matrix	`B = A(colon(1,15));`	`B = A(1:15);`

Table 4-3: Functional Equivalents for the Colon Operator (Continued)

To...	C++ Example	MATLAB Equivalent
Extract elements 1, 3, 5, 7, and 9 from a matrix	`B = A(colon(1,2,10));`	`B = A(1:2:10);`
Create a vector containing the elements 1 through 10	`B = ramp(1,10);`	`B = 1:10;`
Create a vector of odd numbers between 1 and 10	`B = ramp(1,2,10);`	`B = 1:2:10;`

Assignment

You can create scalar arrays using the C++ assignment operator. 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 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";
```

Operators and Function Calls

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;
```

The result, `C`, is equal to `A`.

C++ Constructors and Data Arrays

When you are building large arrays, the `horzcat()` and `vertcat()` functions become cumbersome. You can construct a large array more easily by using an `mwArray` constructor and one or two C++ arrays that contain the data for the array.

When you are initializing an array with a 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`.

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);
```

Both the `mwArray` constructor and the `row2mat()` function take 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`. The entry, “Complex

column-major array of doubles" in the table below demonstrates how to use the fourth argument.

Refer to "Constructors" in Chapter 5 for more information on the `mwArray` constructors.

Summary

The table below summarizes the various ways to construct a new array. Note that the MATLAB and C++ array creation syntax are identical with the exception of the MATLAB : (colon) and [] (bracket) operators.

Table 4-4: Constructing an Array

From...	C++ Example	MATLAB Equivalent	Array Result
Double or integer	`mwArray A = 5;`	`A = 5;`	`5`
Column-major array of doubles	`static double data[] = { 3, 8, 4, 9};` `mwArray A(2, 2, data);`	`A = [3 4 ; 8 9 ];`	`3 4` `8 9`
Row-major array of doubles	`static double data = { 1, 2, 3, 4, 5, 6 };` `mwArray A = row2mat(2, 2, data);`	`A = [ 1 2 3 ; 4 5 6];`	`1 2 3` `4 5 6`

Complex column-major array of doubles	`static double real_data = { 1, 3, 2, 4 };`
	`static double imag_data = { 5, 7, 6, 8 };`
	`mwArray C(2, 2, real_data, imag_data);`
		`A = [ 1 2 ; 3 4 ];`
		`B = [ 5 6; 7 8]*i;`
		`C = A+B;`
			`1+5i  2+6i`
			`3+7i  4+8i`

MATLAB creation function	`mwArray A = ones(4, 4);`
	`mwArray A = eye(3);`
	`mwArray A = rand(6, 9);`
	`mwArray A = magic(10);`
	`mwArray A = zeros(3, 7);`
		`A = ones(4, 4);`
		`A = eye(3);`
		`A = rand(6, 9);`
		`A = magic(10);`
		`A = zeros(3, 7);`
			`4x4 matrix of 1's`
			`3x3 identity matrix`
			`6x9  random matrix`
			`10x10 magic square`
			`3x7 matrix of 0's`

MATLAB colon expression	`mwArray A = ramp(1, 10);` `mwArray A = ramp(3, 2, 17);`	`A = 1:10;` `A = 3:2:17;`	`1 2 3 4 5 6 7 8 9 10` `3 5 7 9 11 13 15 17`
Complex scalar	`mwArray A = complex(3,5);`	`A = 3+5i;`	`A complex number with real part 3 and imaginary part 5.`

Indexing and Subscripts

An indexing expression consists of an array together with one or more subscripts. Applying a subscript to an array allows you to access or change the data stored in the array. The MATLAB C++ Math Library supports both one and two-dimensional indexing. Three-dimensional and higher indexing is not supported.

An array subscript consists of one or two indices. To apply a subscript to an array, you use parentheses placed to the right of the array name. For example, the two-dimensional indexing expression `A(3,1)` returns the element at row three, column one in array `A`. `A(9)`, a one-dimensional indexing expression, returns the ninth element of array `A`. Note that the MATLAB C++ Math Library follows 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," is particularly useful. It 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 ]`.

To modify the data in an array, you use the assignment operator, `=`. For example, the expression `A(3,1) = 45` writes the value `45` into the element at row three, column one of array `A`. If you assign a value to a location that does not exist in the array, the array grows to include that element.

Assigning the null array to an array element deletes the element from the array. For example, `A(3,1) = mwArray();` removes the element at row three, column one. Note that removing an element from an array reshapes the array into a vector.

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.

The next sections show you how to use:

- Two-dimensional subscripts
- One-dimensional subscripts
- Logical subscripts
- An indexing expression in an assignment statement
- A null array to delete elements

Many of the examples use the `horzcat()` and `vertcat()` functions to create the vectors and matrices that are used as indices. `horzcat()` concatenates its arguments horizontally; `vertcat()` concatenates its arguments vertically. You can pass scalar arguments to either function and they will be converted to `mwArrays`. This is true, in general, of MATLAB C++ Math Library routines. See the online *MATLAB C++ Math Library Reference* for further information on the two functions.

TIP: `for`-loops provide an easy way to think 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.

Using Two-Dimensional Subscripts

A two-dimensional subscript contains two indices. The first index is the row index; the second is the column index. Each index can be a scalar, vector, matrix, or 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 two 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: 5 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.

The next section describes how to use two-dimensional indices to select scalars, vectors, and sub-matrices from a matrix. All examples work with example matrix A.

Example Matrix A

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

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 two:

```
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 two:

```
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
```

Large vectors work just as well as the small vectors in these examples. For example, the expression `A(2, horzcat(2,2,2,2,2))` makes five copies of the element at `A(2,2)`.

NOTE: You can pass `horzcat()` up to 32 arguments, any of which can be the result of a nested call to `horzcat()`. `vertcat()` behaves in the same way.

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), and the number of indices in the subscript, `end()` returns the index of the last element in the specified dimension. 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 three:

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

just as

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

does in MATLAB.

The second argument (`2`) to `end()` 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 one and two, 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 one and two:

```
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 two and three:

```
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 three and one, in that order:

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

Subscripts of this form make duplicating the rows or columns of a matrix easy. See Appendix A to learn another technique for duplicating rows and columns.

Selecting an Entire Matrix. Using `colon()` 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.

Using One-Dimensional Subscripts

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 a two-dimensional matrix, you need to know how to go from the one-dimensional index value to a location inside the two-dimensional matrix. A one-dimensional index is like an offset. It tells you how far to count from the beginning of the matrix to reach the element you want.

To count one-dimensionally through a two-dimensional matrix, begin at the first element in the matrix, `(1,1)`, and count down the columns until you have counted up to the index value. When you come to the bottom of a column, continue at the top of the next column.

For example, for the 3-by-3 example matrix `A`,

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

the enumeration is:

```
Column 1:    A(1,1)    A(1)
             A(2,1)    A(2)
             A(3,1)    A(3)

Column 2:    A(1,2)    A(4)
             A(2,2)    A(5)
             A(3,2)    A(6)

Column 3:    A(1,3)    A(7)
             A(2,3)    A(8)
             A(3,3)    A(9)
```

The one-dimensional indexing expression, A(4), accesses the first element in the second column , A(1,2). Its value is 4.

The elements themselves are visited in this order: 1 2 3 4 5 6 7 8 9. Note that matrix A is specially chosen so that A(1) = 1, A(2) = 2, and so on.

The formal rule for a one-dimensional scalar index is: Given an M-by-N matrix R and a scalar integer index X, the one-dimensional indexing expression R(X) selects the element R(row, column), where row equals rem(X–1,M)+1 and column equals ceil(X/M). rem() is the remainder function.

NOTE: The range for a one-dimensional index is from 1, the first element of the matrix, to M*N, the last element in an M-by-N matrix. 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.

The following sections demonstrate how to select a single element with a one-dimensional scalar index, a vector with a one-dimensional vector index, a submatrix with a one-dimensional matrix index, and all elements in the matrix with a one-dimensional colon() index. All examples work with example matrix A.

Example Matrix A

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

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.

Selecting a Single Element

Use a scalar index to select a single element from the matrix. 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 a matrix. 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
```

The `end()` function, which corresponds to the MATLAB `end()` function, provides another method of specifying a vector index. Given an array, a dimension (`1` = row , `2` = column), and the number of indices in the subscript, `end()` returns the index of the last element in the specified dimension.

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 code selects all but the first five elements in matrix `A`:

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

just as

```
A(6:end)
```

does in MATLAB.

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. In the context of an M-by-N matrix `A`, `A(colon())` is always equivalent to `A(transpose(ramp(1,M*N)))`.

Using Logical Subscripts

Logical indexing is a special case of both one- and two-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 a two-dimensional logical subscript by combining a logical index with a scalar, vector, matrix, or `colon()` index.

Selecting from a Matrix

This section demonstrates several ways to use a logical index when selecting elements from a matrix:

- A one-dimensional matrix index
- A pair of logical vector indices for two-dimensional indexing
- A `colon()` index and a logical vector index for two-dimensional indexing

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, if the logical index matrix `B` equals:

```
1 0 1
0 1 0
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 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` equal `[1 0 1]` and `C` equal `[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 one and three.

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

```
4
```

This is tricky. `B`, the row index, selects row one. `C`, the column index, selects column two. There is only one element in array `A` in both row one and column two, 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, `colon()` 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 equal `1 0 1`. Then `A(colon(), B)` equals

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

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

Selecting from a Row or Column

This section demonstrates how to use a logical index to select elements from a row or column.

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 two of matrix X are greater than 9. B is the result of the > 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
```

Using Indexing in Assignment Statements

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 = 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. You are encouraged to experiment with other combinations.

NOTE: When the destination in an assignment statement is an indexing expression, the source and the destination (after the subscript has been applied) must be the same size.

The examples work with matrix A.

Example Matrix A

```
A =

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

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 two and column one to the integer 17. Here, both the source and destination are scalars, and thus the same size.

Assigning to Multiple Elements

Use a vector index to modify multiple elements in a matrix.

colon() frequently appears in the subscript of the destination because it allows you to modify an entire row or column. For example, the expression

```
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 Submatrix

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 one and two: 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)`.

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.

You create a null array with the null `mwArray()` constructor. For example, to delete an element from example matrix A, you assign the null array to that element. For example,

```
A(2,3) = mwArray();
```

deletes the third element in the second row.

When you delete a single element from a matrix, the matrix is converted into a row vector that contains one less element than the original matrix. For example, when element `(2,3)` is deleted from matrix A, 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()) = mwArray();
```

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 the `mwArray()` constructor. 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: A two-dimensional subscript on the left side of the assignment statement can contain only one scalar, vector, or matrix index. The other index used in deletion operations must be a colon index.

C++ and MATLAB Indexing Syntax

The table below summarizes the differences between C++ and MATLAB 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.

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 4-5: MATLAB/C++ Indexing Expression Equivalence

Description	MATLAB Expression	C++ Expression	Result
Extract 1,1 element	`X(1,1)`	`X(1,1)`	`4`
Extract 1st element	`X(1)`	`X(1)`	`4`
Extract 3rd element	`X(3)`	`X(3)`	`5`

Extract all elements into a column vector	`X(:)`	`X(colon())`	`4`
			`6`
			`5`
			`7`

Extract 1st row	`X(1,:)`	`X(1,colon())`	`4 5`
Extract 2nd row	`X(2,:)`	`X(2,colon())`	`6 7`
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 arguments to mwArray::operator() are 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. Of particular note is the default constructor (entry one in the table), which produces an mwIndex object representing : (colon).

Table 4-6: 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 one	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.

Data Conversions

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.

The intelligence with which C++ automatically invokes casts and constructors depends on the compiler you are using. A line of code that the GNU compiler g++ compiles without complaint may cause the SGI compiler to issue an "ambiguous function call" error. The current version of the library compiles without such errors when built with the C++ compilers tested at The MathWorks. However, if you use a different C++ compiler, you may encounter this type of error. To correct the errors, explicitly specify a type with a cast. See the installation guide for a list of supported compilers.

Converting to an mwArray

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, from strings, and from 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. For more information about these data types, see "Data Types" in Chapter 2; for more detail on how to create matrices, refer to the section, "Creating Arrays" in Chapter 4.

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 4-7: 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 from an mwArray

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 4-8: 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;`

mxArray pointer	`const mxArray *ptr;`
	`mwArray A = 5;`
	`ptr = A;`
		`const mxArray *ptr;`
		`mwArray A = 5;`
		`ptr=(const mxArray *)A;`

Refer to “Extracting Data from an mwArray” on page A-7 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.

Array Input and Output

The MATLAB C++ Math Library provides standard C++-style stream input (>>) and output (<<) operators for `mwArray` objects. 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 >>. Though the current >> and << implementations do not read and write MAT-files, you can use the functions `load()` and `save()` discussed below to read and write MAT-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.

In C++, three special input characters describe the shape of the array. The `[` and `]` characters (brackets) enclose an array definition. Within the brackets, the contents of the array appear in row-major order. A semicolon (`;`) separates rows. Note that a semicolon separates rows rather than terminating them. The last row must not end in a semicolon. Array elements may be integers, floating-point numbers, or strings. The input operator ignores space and tab characters between array elements. Note that carriage returns are significant (see below).

In C++, files and the terminal are streams, so that input can be read from and written to both disk files and the user's terminal. There are differences between the type of 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. The only characters permitted between the opening bracket `[` and the closing bracket `]` are the digits 0-9, the period `.` , the minus sign `–` , the semicolon `;` , the letter `e`, which is used for scientific notation, the plus sign + and the letter `i`, which are necessary for matrices containing complex numbers, the `'`, which starts and

ends a string, and the alphabetic characters, which compose a string. Whitespace separates array elements.

Table 4-9: Elements of mwArray Input/Output Syntax

Syntax Element	Definition	Example
[]	Enclose array definition	[1 2]
e	Indicates scientific notation	1e7
.	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'
;	Separates rows	[1 2 ; 3 4]
*	Separates optional scaling factor from array	1e–10 * [1 2]
whitespace	Separates array elements	[1 2]

There is one exception to this format. An optional scaling factor, a double-precision floating-point number, may appear in the input stream before an array. If a scaling factor is present, it applies equally to all of the elements in the array. The scaling factor must be separated from the bracket [that begins the array definition by an asterisk *, denoting multiplication. Use scaling factors when entering very small or very large values.

For convenience, a carriage return may be used in place of a semicolon to separate rows. 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.

Table 4-10: Examples of mwArray Input Syntax

Input	Array	Array Type
`[ 1.4  2.5  3.2 ]`	`1.4  2.5  3.2`	1-by-3 vector
`[ 1 2 ; 3 4 ]`	`1 2` `3 4`	2-by-2 square array
`[ 1 ; 2 3 ]`	Illegal! All rows must be same length.	Invalid array

`[ 1 2`
`  3 4`
`  5 6 ]`
	`1 2`
	`3 4`
	`5 6`
		3-by-2 rectangular array

`[ (1 + 2) 7; 4 5]`	Illegal! Using mathematical expressions in input files is not supported	Invalid array

`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.

`[`
`  1+3i  2+7i ;`
`  9-5i  8+4i`
`]`
	`  1+3i  2+7i`
	`  9-5i  8+4i`
		2-by-2 complex square array

`[`
`  3.14e2  2.73e4;`
`  1.73e3  1.41e2`
`]`
	` 314    27300`
	`1730      141`
		2-by-2 square array. Note use of scientific notation in input.

`'abcd'`	`abcd`	1-by-4 character array. Equivalent to `['abcd']`

Table 4-10: Examples of mwArray Input Syntax (Continued)

Input	Array	Array Type
`[ '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'`.

Example

This simple example reads an array in from standard input and then writes it out to standard output.

```
#include "matlab.hpp"
#include "iostream.h"

int main(int ac, char *av[])
{
    mwArray matrix;

    // Read the matrix in from standard in
    cin >> matrix;

    // Write the matrix out to standard out
    cout << matrix;
}
```

Assume the code above is in a file named `io.cpp`. If the input to this program is stored in a file named `data`, then the program would be built and run as follows on UNIX and PC platforms.

```
mbuild io.cpp
io < data
```

Let the input data file, `data`, contain the following array definition:

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

Running the program yields the following output:

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

Using load( ) and save( )

The MATLAB C++ Math Library provides two functions, load() and save(), that let you import and export array data. save() writes mwArray variables to a MAT-file as named variables; load() reads mwArray variables back in. Since 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.

load() and save() operate on MAT-files. 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, performs the necessary conversion.

Requirements for save( )

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.

Requirements for load( )

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 an example of how to export and import data from your program, see "Example 5: Using load() and save()" in Chapter 3. For more information on MAT-files, consult the online version of the *MATLAB Application Program Interface Guide*.

Output to a GUI

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");
}
```

Setting a Print Handler for 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.

Mathematical Operators

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 4-60.

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 4-11: 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()`

Table 4-11: MATLAB Operator and C++ Function Equivalence (Continued)

Description	Definition: C = A <op> B	MATLAB Operator	C++ Operator	C++ Function
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 4-57 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 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 5 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.

Exception Handling and Error Messages

The MATLAB C++ Math Library delivers error messages via exceptions. This section includes a description of how exception-handling works in C++, how the MATLAB C++ Math Library implements exceptions, and how you can catch the library's exceptions and throw and catch your own exceptions. A list of the exception classes implemented by the library is included at the end of the section. Refer to Appendix C for a list of error messages.

Exception Handling in the MATLAB C++ Math Library

Exceptions are a radical departure from traditional error-handling mechanisms. 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.

Unfortunately, not all C++ compilers fully support exception handling. The MATLAB C++ Math Library, therefore, provides an alternative mechanism that supports exceptions only if your compiler does. 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 current 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 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.

Handling Exceptions in Your Code

To handle the exceptions thrown by the MATLAB C++ Math Library, 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. The MATLAB C++ Math Library provides the functions that help you display the associated messages.

Conceptually, the output from a program written using the MATLAB C++ Math Library divides into two, sometimes three, streams.

1. Ordinary output: text produced by a program in the normal course of its operation.

   The print handling routine determines where this type of text appears. See "Array Input and Output" on page 4-45for more information.

2. Error messages: text produced as a result of error-detecting code that you've written or from exceptions thrown internally by the library.

3. An optional third stream, internal library exception messages: text produced as a result of exceptions thrown internally by the library.

If necessary, you can separate the exceptions thrown internally by the library into a third distinct stream. You need to decide for yourself on a program-by-program basis whether to create a third stream. In most cases, the internal library exception messages appear in the error stream.

This diagram shows the relationships between the error handling functions and the print handler. To follow the possible paths for an error that occurs within your code, start reading the diagram from the "User Error Detection Code" block. To follow the path for an error that occurs within the MATLAB C++ Math Library code, start at the "Internal Library Throw Call" block. The labels “String” and “Exception Object” indicate the type of argument that is passed to the next function.

NOTE: By default, all error messages pass through the print handler. If you want to separate error messages from “ordinary” output, call the function `mwSetErrorMsgHandler()` to replace the default handler, `ErrorMsgHandler()`. If you want to separate exceptions from error messages, call the function `mwSetExceptionMsgHandler()` to replace the `DisplayException()` function.

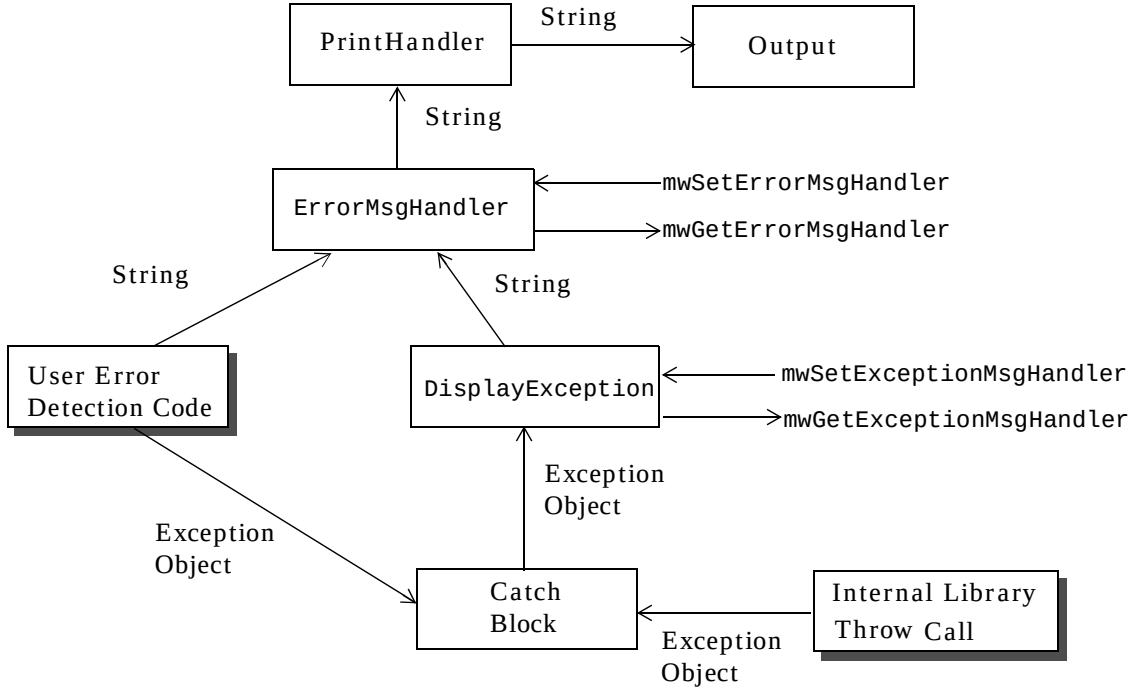


Figure 4-1: Error Handling Functions and the Print Handler

The library contains four functions to help you handle exceptions. The first two functions help you deal with error messages; the second two functions help you handle exceptions.

Using mwSetErrorMsgHandler( ) and mwGetErrorMessageHandler( )

`mwSetErrorMsgHandler()` sets the function used by the library to print error messages. By default, as shown above, the library calls the print handler to print error messages. `mwGetErrorMsgHandler()` returns a pointer to the error message handling function.

```
void mwSetErrorMsgHandler(mwErrorFunc);
mwErrorFunc mwGetErrorMsgHandler();
```

NOTE: Do not call the MATLAB C Math Library function `mlfSetErrorHandler()` function from your application.

Using mwDisplayException( ) and mwSetExceptionMsgHandler( )

`mwDisplayException()` sends an exception to the output function set by the most recent call to `mwSetExceptionMsgHandler()`. If `mwSetExceptionMsgHandler()` has never been called, `mwDisplayException()` uses the default error message handling function or the output function specified by a call to `mwSetErrorMsgHandler()`. If you want a pointer to the default error message handling routine, you can call `mwGetErrorMsgHandler()`.

Note that `mwSetExceptionMsgHandler()` takes an `mwExceptionMsgFunc` argument rather than an `mwOutputFunc` argument.

```
void mwDisplayException(const mwException &ex);
void mwSetExceptionMsgHandler(mwExceptionMsgFunc f);
```

mwErrorFunc vs. mwExceptionMsgFunc

The difference between these two function types is in their arguments. An `mwErrorFunc` takes a `const char *` (the message to display) and a Boolean value (true for an error, false for a warning) and returns `void`, having displayed the message. An `mwExceptionMsgFunc` takes a `const mwException&` (the exception to display) and returns `void`, having first rendered the exception into

a string and then having sent the string to an output function (at least, that's what the default implementation does).

```
typedef void (*mwErrorFunc)(const char *, mwBool);
typedef void (*mwExceptionMsgFunc)(const mwException &);
```

You are more likely to write an `mwErrorFunc`, a function which separates error messages from ordinary output, than an `mwExceptionMsgFunc`, a function which separates exceptions from error messages. Most applications do not require both a separate exception stream and a separate error message stream. However, many applications do separate error messages from ordinary output.

A Simple Try-Block and Catch-Block

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.

NOTE: You should always have a try and catch block like this one in your main routine. 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 the error messages associated with the exceptions thrown by the library.

Redirecting Errors to the Standard Error Stream

The above example uses the default exception handling routine, which sends all output to the standard output stream, cout. This output usually appears directly on the user's screen. The slightly more complex example below uses mwSetErrorMsgHandler() to redirect all errors to the standard error stream. Note that the call to mwDisplayException() in the catch-block does not change

from the previous example. By default, `mwDisplayException()` calls the error handler set by `mwSetErrorMsgHandler()`.

```
// New error message handler
void ToStandardError(const char *msg, mwBool isError)
{
    if(isError)
        cerr << msg << endl;
    else
        cerr << "WARNING: " << msg << endl;
}

// try-block
try
{
     mwSetErrorMsgHandler(ToStandardError);
     eig(A);
}

// catch-block
catch(mwException &ex)
{
    mwDisplayException(ex);
}
```

The Library's Default Handlers

`mwDisplayException()` does not have to use the error message handler set by `mwSetErrorMsgHandler()`. The function `mwSetExceptionMsgHandler()` sets the function used to display the messages associated with exceptions. For example, here's the default exception message handler's implementation:

```
// MATLAB C++ Math Library's default exception message handler
static void DefaultException(const mwException& ex)
{
      strstream os;
      os << ex << endl << ends;
      (*ErrorMsgHandler)(os.str(), true);
}
```

The default exception message handler converts the error message associated with the exception into a string, using the `strstream` (string stream) object, and then sends that string out via the error message handling function.

For comparison, here's the implementation of the default error message handler:

```
// MATLAB C++ Math Library's default error message handler
static void DefaultError(const char *msg, mwBool isError)
{
    strstream os;

    if (isError)
        os << msg << endl << ends;
    else
        os << "WARNING: " << msg << endl << ends;

    if (PrintHandler != 0)
        (*PrintHandler)(os.str());
}
```

The default error message handler simply calls the default print handler, which sends the message to standard output.

```
//MATLAB C++ Math Library's default print handler
static void DefaultOutput(const char *msg)
{
    cout << msg;
}
```

Other implementations, such as logging error messages in a file or displaying dialog boxes in a graphical user interface, are also possible. See "Output to a GUI" on page 4-52 for details on interacting with a graphical user interface.

Exception Classes

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`.

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 `stdexcept.h` in the `extern/include/cpp` directory of your installation.

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 your 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.

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.

For efficiency, the pool is subdivided into several smaller pools, each of which contains a list of available memory blocks. All of the blocks in a given pool are the same size; generally, the sizes are a power of two. After a certain point, rounding up to the nearest power of two becomes wasteful, and the memory allocator returns blocks of exactly the requested size.

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,
                           mwMemCompactFunc = 0);
```

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);
typedef void (*mwMemCompactFunc)(void);
```

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. However, the last argument to `mwSetLibraryAllocFcns()`, `mwMemCompactFunc`, is not currently used and is therefore initialized to zero. When you call `mwSetLibraryAllocFcns()`, you do not need to specify a value for it.

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.

To illustrate how the library implements its memory management routines and how you need to structure your routines, here are the default memory management routines from the MATLAB C++ Math Library.

Calloc Allocation Routine

`DefaultCalloc()` is the default calloc memory allocation routine for the MATLAB C++ Math Library. Any memory calloc routine that you write must have the same form as `DefaultCalloc()`, which conforms to the type:

```
typedef void *(*mwMemCallocFunc)(size_t, size_t);
```

`DefaultCalloc()` allocates a block of memory based on the number of contiguous elements that you want allocated (`count`) and an integer

representing the size of each element (size). The routine initializes the allocated memory to zero.

```
static void *DefaultCalloc(size_t count, size_t size)
{
    if (memory_pool==0)
        memory_pool = new mwMultiBlockMgr;

    int bytes = count * size;
    void *data = memory_pool->New(bytes);

    if (data)
        memset(data, 0, bytes);

    return data;
}
```

Deallocation Routine

If you write a memory allocation routine, you must write a corresponding routine that frees memory. `DefaultFree()` is the default deallocation routine for the MATLAB C++ Math Library. Any memory free routine that you write must have the same form as `DefaultFree()`, which conforms to this type:

```
typedef void (*mwMemFreeFunc)(void *);
```

`DefaultFree()` takes a pointer to the beginning of the memory block to be freed and returns void.

```
static void DefaultFree(void *ptr)
{
    if (memory_pool==0)
     MLM_THROW1(mwLogicError, "No memory pools. Can't free %x.", \
                 ptr)
    memory_pool->Delete((char *)ptr);
}
```

Note that `DefaultFree()` throws an exception if the program attempts to return memory before allocating memory. You may want to take similar precautions.

The overloaded `delete` operator in `mwArray` calls this function, as does `mxFree()`.

Reallocation Routine

`DefaultRealloc()` is the default reallocation routine for the MATLAB C++ Math Library. Any memory reallocation routine that you write must have the same form as `DefaultRealloc()`, which conforms to this type:

```
typedef void *(*mwMemReallocFunc)(void *, size_t);
```

`DefaultRealloc()` 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 *DefaultRealloc(void *ptr, size_t size)
{
    if (memory_pool==0)
        MLM_THROW1(mwLogicError, "No memory pools.  \
                   Can't realloc %x.", ptr)
    return memory_pool->Realloc((char *)ptr, size);
}
```

Malloc Allocation Routine

`DefaultMalloc()` is the default allocation routine for the MATLAB C++ Math Library. Any memory allocation routine that you write must have the same form as `DefaultMalloc()`, which conforms to this type:

```
typedef void *(*mwMemAllocFunc)(size_t);
```

The default memory allocation routine takes an integer size that represents the number of bytes to allocate and returns a pointer to `void`. Unlike `DefaultCalloc()`, `DefaultMalloc()` does not initialize the memory it returns.

```
static void *DefaultMalloc(size_t size)
{
    if (memory_pool==0)
        memory_pool = new mwMultiBlockMgr;
    return memory_pool->New(size);
}
```

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.

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 4-12: 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.

5

Library Routines

This chapter is a reference guide for the `mwArray` class, of operators that you use with arrays, and the more than 350 functions contained in the MATLAB C++ Math Library.

The chapter consists of five sections:

- `mwArray` Class Interface
- 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 *C++ Math Library Reference* for a complete definition of the function syntax and arguments.

mwArray Class Interface

The mwArray class public interface (those functions you can call directly) is relatively small, consisting of constructors and destructor, overloaded new and delete operators, one user-defined conversion, four indexing operators, 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 350 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 * pointer, or another mwArray object. This table lists the most commonly used constructors.

mwArray Constructors

Constructor	Creates	Example
`mwArray()`	NULL 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(int, int, int)`	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()`

  Make a NULL array, an array with no contents.

- `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 "Creating Arrays" in Chapter 4 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. This constructor is slightly more efficient than the previous one, because it uses integers rather than double-precision floating-point numbers.

- `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 "Creating Arrays" in Chapter 4 for more examples of how to use constructors.

Indexing and Subscripts

Indexing is implemented through the complex interaction of three classes: `mwArray`, `mwSubArray`, and `mwIndex`. The indexing operator is `()`, and its usual argument is an `mwIndex`, which can be made from a scalar or another array. When applied to an `mwArray`, `operator()` returns an `mwSubArray`. The `mwSubArray` "remembers" the indexing operation; it defers evaluation of it until the result is either assigned or referred to.

The MATLAB C++ Math Library supports one and two-dimensional indexing.

- `mwSubArray operator()(const mwIndex &a) const`

  This routine implements one-dimensional indexing with an `mwIndex` object providing the subscript.

- `mwSubArray operator()(const mwIndex &a)`

  This routine modifies the contents of an array using one-dimensional indexing. Because this routine is non-`const`, calls to it are valid targets for the assignment operator.

- `mwSubArray operator()(const mwIndex &a, const mwIndex &b) const`

  This is the most general form of two-dimensional indexing. Because `mwIndex` objects can be made from integers, double-precision floating-point numbers and even `mwArrays`, this routine can handle two-dimensional indexing of any type.
- `mwSubArray operator()(const mwIndex &a, const mwIndex &b)`

  Like its one-dimensional counterpart, this routine allows two-dimensional indexing expressions as the target of assignment statements.

See "Indexing and Subscripts" in Chapter 4 for examples of how these routines are used.

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 4 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. "Array Input and Output" in Chapter 4 describes the syntax of the input format.

The stream operators call `Read()` and `Write()`, `mwArray` public member functions.

- `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 `]`. "Array Input and Output" in Chapter 4 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.

NOTE: `Write()` writes arrays in exactly the format that `Read()` reads them. An array written by `Write()` can be read by `Read()`.

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 5-4 above), 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. In this version of the library, this function always returns two.

- `int32 Size(int32 dim) const`

  Return the size (number of elements) of the indicated dimension. The integer argument to this function must be either 1 or 2.

- `int32 Size(int32* dims) const`

  Determine the sizes of all the dimensions of the array and return them via the given integer array. The input integer array must contain enough space to store at least two integers. This function's return value is the number of dimensions of the array; that number is always 2.

For example, this code demonstrates the difference in efficiency between one of the mwArray Size member functions and the non-member 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);
```

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 matrix is the result of applying the operation to the corresponding elements of the operand matrices. 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.

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.

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.

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.

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()

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 350 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 *C++ Math Library Reference*. Also refer to the section "Calling Conventions" in Chapter 4 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 non-zero.
and_func	Logical AND (&).
any	True if any element of vector is non-zero.
not_func	Logical NOT (~).
or_func	Logical OR (|).
xor_func	Logical exclusive-or operation.

Set Operators

Function	Purpose
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.

MATLAB as a Programming Language

Function	Purpose
`feval`	Function evaluation.
`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
`automesh`	True if the inputs require automatic meshgriding.
`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`, `rand_func`	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.

Basic Array Information (Continued)

Function	Purpose
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....
`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.

Specialized Matrices (Continued)

Function	Purpose
`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.
fftshift	Move zeroth lag to center of spectrum.
ifft	Inverse discrete Fourier transform.
ifft2	Two-dimensional inverse discrete 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.

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.
`strjust`	Justify a character array.
`strncmp`	Compare the first n characters of two strings.
`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.
`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.
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	Is file position indicator at the end of the file?
ferror	Inquire file I/O error status.
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.

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 (*mwMemFreeFunc)(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 “Exception Handling and Error Messages” in Chapter 4; for more information on print handling , see “Array Input and Output” in Chapter 4.

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.

The section "Indexing and Subscripts" in Chapter 4, 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 Application Program Interface 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 Application Program Interface 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.
`mxCreateCharArray`	Create an unpopulated N-dimensional string `mxArray`.
`mxCreateCharMatrixFromStrings`	Create a populated 2-dimensional string `mxArray`.
`mxCreateDoubleMatrix`	Create an unpopulated 2-dimensional, double-precision, floating-point `mxArray`.
`mxCreateNumericArray`	Create an unpopulated N-dimensional numeric `mxArray`.
`mxCreateString`	Create a 1-by-n string `mxArray` initialized to the specified string.
`mxDestroyArray`	Free dynamic memory allocated by an `mxCreate` routine.
`mxDuplicateArray`	Make a deep copy of an array.
`mxGetClassID`	Get (as an enumerated constant) an `mxArray`'s class.

Array Access Routines (Continued)

Function	Purpose
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.
mxGetImagData	Get pointer to imaginary data of an mxArray.
mxGetInf	Get the value of infinity.
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.
mxGetPi, mxGetPr	Get the real and imaginary parts of a double mxArray.
mxGetScalar	Get the real component of an mxArray's first data element.
mxGetString	Copy a string mxArray's data into a C-style string.
mxIsChar	True if a string mxArray.
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.

Array Access Routines (Continued)

Function	Purpose
mxIsInf	True if value is infinite.
mxIsInt8	True if mxArray represents its data as signed 8-bit integers.
mxIsInt16	True if mxArray represents its data as signed 16-bit integers.
mxIsInt32	True if mxArray represents its data as signed 32-bit integers.
mxIsLogical	True if mxArray is Boolean.
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.
mxIsString	Inquire if a matrix contains string data.
mxIsUint8	True if mxArray represents its data as unsigned 8-bit integers.
mxIsUint16	True if mxArray represents its data as unsigned 16-bit integers.
mxIsUint32	True if mxArray represents its data as unsigned 32-bit integers.
mxMalloc	Allocate dynamic memory using MATLAB's memory manager.
mxRealloc	Reallocate memory.
mxSetData	Set pointer to data.

Array Access Routines (Continued)

Function	Purpose
mxSetDimensions	Modify the number of dimensions and/or the size of each dimension.
mxSetImagData	Set imaginary data pointer for an mxArray.
mxSetLogical	Set the logical flag.
mxSetM, mxSetN	Set the number of rows (M) and columns (N) of an array.
mxSetPi, mxSetPr	Set the real and imaginary parts of a double mxArray.

Directory Organization

This chapter describes the directory organization of the MATLAB C++ Math Library on UNIX and Microsoft Windows systems. The C++ Math Library is not available for Macintosh computers.

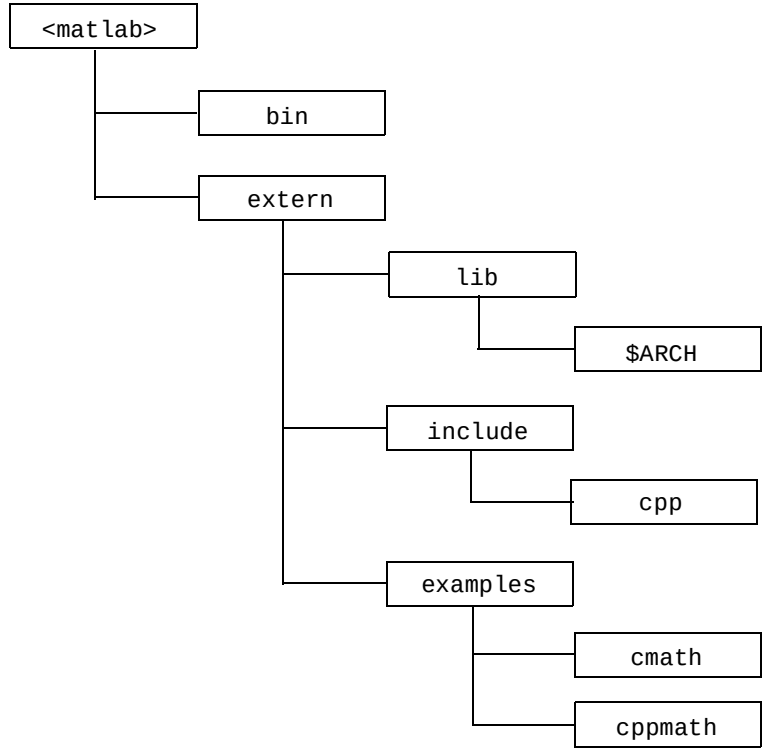
Refer to this chapter to find out what files the MATLAB C++ Math Library installs on your computer, what the purpose of each file is, and where each file is stored. For instructions on how to install the software, see “Installing the C++ Math Library” in Chapter 1.

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 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 SunOs 4.x, sun4 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.
`libmcc.`*ext*	MATLAB Compiler Library for stand-alone applications. Contains the `mcc` and `mcm` routines required for building stand-alone applications.
`libmi.`*ext*	Internal math 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 Application Program Interface Library. Contains array access routines.
`libut.`*ext*	MATLAB Utilities Library. Contains the utility routines used by various components.

where `.`*ext* is

`.a` on IBM RS/6000 and Sun4; `.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.

`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.

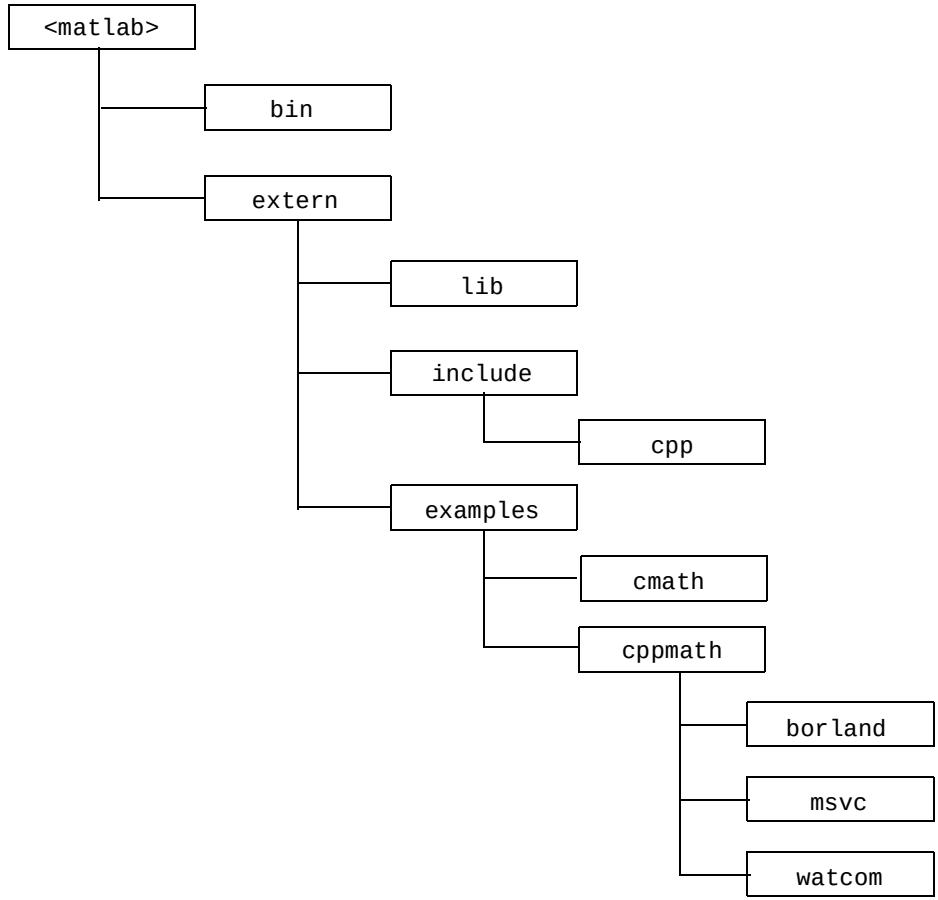
<matlab>/extern/examples/cppmath

The `<matlab>/extern/examples/cppmath` directory holds the sample C++ programs described in the Chapter 3. An additional subdirectory, `example9`, contains the M-files associated with "Example 9: Using the MATLAB Compiler" on page 3-49.

`ex1.cpp`	The source code for "Example 1: Creating Arrays and Array I/O" on page 3-3.
`ex2.cpp`	The source code for "Example 2: Writing Simple Functions" on page 3-7.
`ex3.cpp`	The source code for "Example 3: Calling Library Routines" on page 3-10.
`ex4.cpp`	The source code for "Example 4: Handling Errors" on page 3-15.
`ex5.cpp`	The source code for "Example 5: Using load() and save()" on page 3-19.
`ex6.cpp`	The source code for "Example 6: Using File I/O Functions" on page 3-23.
`ex7.cpp`	The source code for "Example 7: Rewriting roots.m in C++" on page 3-28.
`ex8.cpp`	The source code for "Example 8: Passing Functions As Arguments" on page 3-36.
`ex9.cpp`	The source code for "Example 9: Using the MATLAB Compiler" on page 3-49.
`edge.cpp`	Edge detection code used in "Example 9: Using the MATLAB Compiler" on page 3-49.
`release.txt`	Release notes for the current release of the MATLAB C++ Math Library.
`trees.bmp`	Bitmap file used in "Example 9: Using the MATLAB Compiler" on page 3-49.

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.
libmcc.dll	MATLAB Compiler Library for stand-alone applications. Contains the mcc and mcm routines required for building stand-alone applications.
libmi.dll	Internal math 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 Application Program Interface 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	Switches and settings for the C++ compiler to create stand-alone applications, e.g., msvccompp.bat for use with Microsoft Visual 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	MATLAB C++ Math Library for the Borland C++ compiler, v5.0 and v5.2.
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.

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.
libmcc.def _libmcc.def	Contains names of functions exported from the MATLAB Compiler Library DLL for stand-alone applications.
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.
libut.def _libut.def	Contains names of functions exported from libut.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 described in the Chapter 3. An additional subdirectory,

example9, contains the M-files associated with "Example 9: Using the MATLAB Compiler" on page 3-49.

ex1.cpp	The source code for "Example 1: Creating Arrays and Array I/O" on page 3-3.
ex2.cpp	The source code for "Example 2: Writing Simple Functions" on page 3-7.
ex3.cpp	The source code for "Example 3: Calling Library Routines" on page 3-10.
ex4.cpp	The source code for "Example 4: Handling Errors" on page 3-15.
ex5.cpp	The source code for "Example 5: Using load() and save()" on page 3-19.
ex6.cpp	The source code for "Example 6: Using File I/O Functions" on page 3-23.
ex7.cpp	The source code for "Example 7: Rewriting roots.m in C++" on page 3-28.
ex8.cpp	The source code for "Example 8: Passing Functions As Arguments" on page 3-36.
ex9.cpp	The source code for "Example 9: Using the MATLAB Compiler" on page 3-49.
edge.cpp	Edge detection source code, used in "Example 9: Using the MATLAB Compiler" on page 3-49.
release.txt	Release notes for the current release of the MATLAB C++ Math Library.
trees.bmp	Bitmap file used in "Example 9: Using the MATLAB Compiler" on page 3-49.

<matlab>\extern\examples\cppmath\borland

The <matlab>\extern\examples\cmath\borland directory contains Borland C++ v5.0 project files for each of the MATLAB C++ Math Library examples.

ex*.ide	Borland C++ project files for each MATLAB C++ Math Library example.

<matlab>\extern\examples\cppmath\msvc

The <matlab>\extern\examples\cmath\msvc directory contains Microsoft Visual C++ v4.2 project files and associated makefiles for each of the MATLAB C++ Math Library examples.

ex*.mak ex*.mdp	Microsoft Visual C++ project files for each MATLAB C++ Math Library example.

<matlab>\extern\examples\cppmath\watcom

The <matlab>\extern\examples\cmath\watcom directory contains Watcom C++ v10.6 project files and associated makefiles for each of the MATLAB C++ Math Library examples.

ex*.mk ex*.mk1 ex*.tgt ex*.wpj	Watcom C++ project files for each MATLAB C++ Math Library example.

A

Tricks, Tips, and Techniques

This appendix presents several one-page problem statements and solutions. In general, the solutions are presented as code fragments that you can copy directly into your own programs. The examples are much shorter than those presented in Chapter 3 and contain techniques that experienced programmers and MATLAB users will find useful.

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 submatrix 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.

Applying a Subscript to the Result of an Indexing Operation

You can stack subscripts to the right of an `mwArray` object; they'll be executed in left-to-right order. For example, this code extracts the fourth row of a 10-by-10 array, `A`, and turns it into a column vector:

```
mwArray A = magic(10), B;
B = (A(4,colon()))(colon());
```

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
```

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*, Appendix B, or the header file `matrix.h`.

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

int *integers = new int[ 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;
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.

Array Size

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

- `size()`, the C++ Math Library version of the MATLAB `size()` function, which returns the number of rows and columns in an array
- an overloaded version of `size()` that returns integers
- the `mwArray` `Size()` member functions
- the `mwArray` `EltCount()` member function

Most MATLAB users instinctively use `size()`. However, `size()` returns one or more scalar arrays, which, while consistent with the rest of the interface, requires an unnecessary amount of time and space. Because the overloaded version of `size()` returns integers rather than scalar arrays, it is more efficient than the first `size()` and just as easy to use. The `mwArray` member functions are the most efficient way to compute the dimensions of an array.

size()

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

This version of `size()` returns the number of rows and columns in an array, but is far slower and uses more space than the overloaded version of `size()` below.

Overloaded size()

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

This version produces the same result, but here, `m` is set to `4` and `n` to `7`.

mwArray Size() Member Functions

Careful reading of the `mwArray` header file reveals another set of size functions. These `mwArray` member functions are the most efficient way to compute the

dimensions of an array. Using array A from the examples above, you can use these functions in various ways:

```
int32 dims[2];
A.Size(dims);        // Sets dims to (4, 7)
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

A final function, `EltCount()`, 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.

Shrinking an mwArray

In MATLAB, you can shrink an array by assigning the null array to one or more of the array elements. The MATLAB C++ Math Library emulates this behavior. You remove elements from an `mwArray` by assigning the null array to the elements you want to remove: a single element, a group of elements, or an entire row or column. Shrinking an array by removing a single element or group of elements turns the array into a vector; removing an entire row or column simply reduces one of the array dimensions by one. See "Indexing and Subscripts" in Chapter 4 and "Constructors" in Chapter 5 for more information on indexing and the null array.

Removing a Single Element

Removing a single element from an array turns it into a vector. For example, removing the (2,2) element from a 4-by-4 array turns it into a 15-element vector. The vector is formed from the columns of the array, i.e., the first four elements in the vector are the elements in the first column of the array, etc.

```
mwArray A = magic(4);
A(2,2) = mwArray();
```

The (2,2) element of the array is not present in the resulting vector.

Removing a Group of Elements

You can remove a group of elements from an array by specifying a nonscalar subscript on the left-hand side of the assignment statement. In this case, two vectors, a 1-by-5 vector and a 1-by-4 vector, determine the elements that are removed from array `A`, a 12-by-12 array of random numbers.

```
mwArray A = rand(12);
A(ramp(3,7), ramp(9,12)) = mwArray();
```

This code removes elements (3,9) through (3,12), then (4,9) through (4,12), etc., stopping once it has removed (7,9) through (7,12). What was once a 12-by-12 array is now a 1-by-124 (144 - 20, or (12*12) - (5*4)) vector.

Removing an Entire Row or Column

Conceptually, this is the simplest of the three types of array-shrinking operations. The array remains a matrix but gets smaller.

```
mwArray A = magic(8);
A(3,colon()) = mwArray();  // Removes the 3rd row
A(colon(),5) = mwArray();  // Removes the 5th column
```

After the first shrinking, the magic square becomes a nonmagic rectangle; after the second statement it is square again, (7-by-7), but remains nonmagical.

Using Streams

You can play many interesting tricks with stream input and output, particularly when you consider that streams can be made from both strings and files. Some of the examples presented below are more hints than code fragments. The details of the ideas they describe are beyond the scope of this document.

File Input and Output

The simplest application of streams is file input and output. "Example 1: Creating Arrays and Array I/O" in Chapter 3 demonstrates some simple input from and output to the terminal. Reading or writing arrays from and to files isn't much more difficult. Use the C++ class `ifstream` to create file input streams and `ofstream` to create file output streams.

This code writes array `A` to a file and then reads the data from the file into array `B`:

```
mwArray A = rand(99), 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.

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`.

For more information on the `MLM_TRHROW` macros, see "Using the MLM_THROW Macros to Throw Exceptions" in Chapter 4.

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 `strstream` class demonstrated above, 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 subclass `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. Subclassing the `istream` and `ostream` classes, however, is a difficult task.

B

The mxArray Structure

Each `mwArray` object contains a pointer to an instance of MATLAB's most important data structure: the `mxArray`. This appendix describes the `mxArray` structure in enough detail to allow you to perform the most common operations on it. For complete details, see the *MATLAB Application Program Interface Guide* and the *MATLAB C Math Library* manual.

The Matrix

A matrix is a two-dimensional array of double-precision floating-point numbers. Matrix data is either *real* (one number per element) or *complex* (two numbers per element). A matrix may also be either *full* or *sparse*; in a full M-by-N matrix, enough space is allocated to store all of the M*N elements, while a sparse matrix only allocates enough space to store the nonzero elements. The space savings afforded by a sparse matrix is offset by a higher runtime cost for accessing an element.

Conceptually, a full matrix consists of four fields:

- The number of rows, usually denoted by `M`.
- The number of columns, or `N`.
- A pointer to the real part of the data, often abbreviated to `pr`.
- A pointer to the imaginary part of the data, often abbreviated to `pi`.

A sparse matrix has four additional fields:

- The maximum number of nonzero elements in the matrix, or `nzmax`.
- The actual number of nonzero elements in the matrix, or `nnz`.
- The row indices of the nonzero elements, or `ir`.
- The column index information, or `jc`. For 0 <= j <= N-1, jc[j] is the index into the list of row indices of the first nonzero element in the jth column and jc[j+1]-1 is the index of the last nonzero entry in that column.

NOTE: The MATLAB C Math Library does not currently support sparse matrices.

Accessing Matrix Fields

You can access the fields of a matrix via function calls. For each field there are two functions, one for reading the data and one for writing it. For each field, the name of the field determines the name of the function used to access that field. Access function names have three parts. Each function name begins with the prefix `mx`. Next comes the action, either `Set` or `Get`. The third and final component of the function name is the field name, for example `Pr` or `Nnz`. So, to

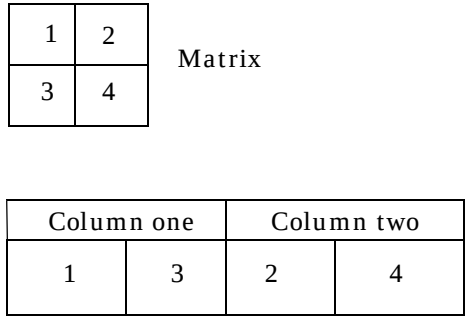
determine the number of columns in a matrix, call `mxGetN()`, and to return a pointer to the complex part of a complex matrix, call `mxGetPi()`. We do not recommend calling the set routines on any `mxArray*` that you obtain via `GetData()` from an `mwArray` object. Unless you are very careful, you risk corrupting the data structure and confusing the `mwArray` object in the process.

Matrix Data Storage Layout

The numerical data stored in the matrix is stored in column-major order whereas C and C++ store their arrays in row-major order. If you're a C or C++ programmer, be aware that your intuitive notion of the way to iterate over the elements in a MATLAB matrix is probably backwards. You don't need to worry about this difference when you use the indexing operators in the MATLAB C++ Math Library, but you must take it into account when you examine the raw data.

For example, consider the matrix `[ 1 2 ; 3 4 ]`, which has two rows and two columns. The first row consists of the numbers `1` and `2`, the second row, the numbers `3` and `4`. The first column consists of the numbers `1` and `3`, the second column, the numbers *2* and `4`. This matrix is stored in memory in column-major order: all the elements of column one are followed by all the elements of column two. In a column-major storage scheme, the elements of a given column are contiguous and the beginning of each column is separated from the beginning of the next by a number of elements equal to the number of rows in the matrix.

Figure 2-1: Matrix Storage Layout



C

Error Messages

Error Messages

This appendix 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.

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.

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

`<number> indices: exceeding max. number of indices: <number>.`

This error indicates an attempt to index a matrix with more than two indices. You should never see this error because code like `A(1,2,3)` will not compile. If you do, please report it to The MathWorks.

`Attempt to remove a key that doesn't exist.`

This internal error indicates that the memory manager's data structures are corrupt. This error can be caused by memory bugs in the MATLAB C++ Math Library or in your own code. Use Purify or BoundsChecker to discover when and where memory is being corrupted. Also see "Memory Management" in Chapter 4.

`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 8: Passing Functions As Arguments" in Chapter 3 for more details on using `FreezeData()`.

`Corrupt memory pool data. Block size = <number>.`

This is an internal error. It indicates that the data structures used by the memory manager are corrupt. This error can be caused by a bug in your code or one in the library itself. If you have access to a memory analysis tool like Purify or BoundsChecker, try using it to determine where the problem occurs.

`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.

`Failed to allocate <number> bytes.`

The memory manager could not allocate the requested amount of memory. This means that either the memory manager data structures are corrupt, or the program is running out of memory. If you have access to a memory analysis tool like Purify or BoundsChecker, try using it to determine where the problem occurs.

`Hash table allocation failed.`

The initial attempt to allocate a hash table failed. This may be a sign that your program is running out of memory. Try reducing the amount of memory in use by closing programs that you aren't actively using.

`Hash table size (<number>) invalid.`

This internal error indicates that a hash table data structure is corrupt; the table size is larger than it was expected to be. This usually means that some code somewhere has written into memory that it doesn't own.

`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.

`Index (%d) out of range (<number>:<number>).`

The index you specified does not reference a valid matrix element. This error most often occurs when reading from a matrix, since writing to an element that does not exist will usually cause the matrix to grow to accommodate that element. However, this error can occur during writing or reading if you specify a completely invalid index, for example, one that is less than the index base, the minimum legal index.

`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.

`Just grew hash table, but no space for new key.`

This internal error indicates an error in the hash table code. Hash tables are used extensively in the memory management subsystem.

`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.

`Memory block of size <number> starting at <hex number> allocated more than once.`

This error indicates that the memory manager has given the same block of memory out in response to two or more different memory allocation

requests. The memory manager's data structures have become corrupted. If you have access to a memory analysis tool like Purify or BoundsChecker, try using it to determine where the problem occurs.

`Need array pointer to determine size of ':'.`

This is an internal error that indicates the `mwArray` object is corrupt.

`No memory manager for blocks of size %d, cannot return to pool.`

The memory manager maintains several pools of free memory. All the elements in a given pool must be the same size. This error occurs when an element is the wrong size for the pool to which it is being returned. If you have access to a memory analysis tool like Purify or BoundsChecker, try running it to determine where the error is occurring.

`No memory pools. Can't free <address>.`

The program tried to return a block of memory before any blocks of memory were allocated. This may be caused by an internal error or by a bug in your program.

`Non-empty pools can't grow.`

This internal error indicates a problem with the memory manager. The error occurs when the memory manager tries to increase the size of a memory pool that is not empty. This error does NOT mean that your program is out of memory. If you have access to a memory analysis tool like Purify or BoundsChecker, try running it to determine where the error is occurring.

`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)`.

`Size of returned memory block (<number>) is not a multiple of <number>.`

All the memory blocks must be aligned on a machine-specific basis (usually on 64-bit boundaries). This error occurs when the memory management subsystem detects that a block is not the correct size. If you have access to a memory analysis tool like Purify or BoundsChecker, try running it to determine where the error is occurring.

`Size of the memory pool (<number>) was not a multiple of <number>.`

This is an internal error, indicating a problem with the memory management subsystem. Bugs in your code or the library can cause this error. If you have access to a memory analysis tool like Purify or BoundsChecker, try running it to determine where the error is occurring.

Index

Symbols

A

D

E

F

G

K

L

M

N

O

Q

R

S

T

U

V

W

X

Z