

**MATLAB® Compiler SDK™**

C/C++ User's Guide



**MATLAB®**

R2022b



## How to Contact MathWorks



Latest news: [www.mathworks.com](http://www.mathworks.com)  
Sales and services: [www.mathworks.com/sales\\_and\\_services](http://www.mathworks.com/sales_and_services)  
User community: [www.mathworks.com/matlabcentral](http://www.mathworks.com/matlabcentral)  
Technical support: [www.mathworks.com/support/contact\\_us](http://www.mathworks.com/support/contact_us)



Phone: 508-647-7000



The MathWorks, Inc.  
1 Apple Hill Drive  
Natick, MA 01760-2098

*MATLAB® Compiler SDK™ C/C++ User's Guide*

© COPYRIGHT 2012–2022 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

### Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See [www.mathworks.com/trademarks](http://www.mathworks.com/trademarks) for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

### Patents

MathWorks products are protected by one or more U.S. patents. Please see [www.mathworks.com/patents](http://www.mathworks.com/patents) for more information.

### Revision History

March 2015	Online only	New for Version 6.0 (Release R2015a)
September 2015	Online only	Revised for Version 6.1 (Release 2015b)
October 2015	Online only	Rereleased for Version 6.0.1 (Release 2015aSP1)
March 2016	Online only	Revised for Version 6.2 (Release 2016a)
September 2016	Online only	Revised for Version 6.3 (Release R2016b)
March 2017	Online only	Revised for Version 6.3.1 (Release R2017a)
September 2017	Online only	Revised for Version 6.4 (Release R2017b)
March 2018	Online only	Revised for Version 6.5 (Release R2018a)
September 2018	Online only	Revised for Version 6.6 (Release R2018b)
March 2019	Online only	Revised for Version 6.6.1 (Release R2019a)
September 2019	Online only	Revised for Version 6.7 (Release R2019b)
March 2020	Online only	Revised for Version 6.8 (Release R2020a)
September 2020	Online only	Revised for Version 6.9 (Release R2020b)
March 2021	Online only	Revised for Version 6.10 (Release R2021a)
September 2021	Online only	Revised for Version 6.11 (Release R2021b)
March 2022	Online only	Revised for Version 7.0 (Release R2022a)
September 2022	Online only	Revised for Version 7.1 (Release R2022b)

## Installation and Configuration

**1**

<b>Configure the mbuild Options File</b> .....	<b>1-2</b>
<b>Solve Installation Problems</b> .....	<b>1-3</b>

## Libraries

**2**

<b>Call a C Shared Library</b> .....	<b>2-2</b>
Restrictions When Using MATLAB Function loadlibrary .....	<b>2-5</b>
<b>Integrate C++ Shared Libraries with mxArray</b> .....	<b>2-6</b>
C++ Shared Library Wrapper .....	<b>2-6</b>
C++ libmatrix Example .....	<b>2-6</b>
<b>Use Multiple Shared Libraries in Single Application</b> .....	<b>2-9</b>
Initialize and Terminate Multiple Shared Libraries .....	<b>2-9</b>
Work with MATLAB Function Handles .....	<b>2-10</b>
Work with Objects .....	<b>2-13</b>
<b>Understand the mclmcrtr Proxy Layer</b> .....	<b>2-16</b>
<b>Call MATLAB Compiler SDK API Functions from C/C++</b> .....	<b>2-17</b>
Shared Library Functions .....	<b>2-17</b>
Type of Application .....	<b>2-17</b>
Structure of Programs That Call Shared Libraries .....	<b>2-18</b>
Library Initialization and Termination Functions .....	<b>2-18</b>
Print and Error Handling Functions .....	<b>2-19</b>
Functions Generated from MATLAB Files .....	<b>2-20</b>
Retrieving MATLAB Runtime State Information While Using Shared Libraries .....	<b>2-24</b>
<b>Memory Management and Cleanup</b> .....	<b>2-25</b>
Overview .....	<b>2-25</b>
Passing mxArray to Shared Libraries .....	<b>2-25</b>
<b>Write Applications for macOS</b> .....	<b>2-26</b>
Objective-C/C++ Applications for Apple's Cocoa API .....	<b>2-26</b>
Where's the Example Code? .....	<b>2-26</b>
Preparing Your Apple Xcode Development Environment .....	<b>2-26</b>
Build and Run the Sierpinski Application .....	<b>2-27</b>
Running the Sierpinski Application .....	<b>2-28</b>

## Deployment Process

### 3

<b>About the MATLAB Runtime</b> .....	3-2
How is MATLAB Runtime Different from MATLAB? .....	3-2
Performance Considerations for MATLAB Runtime .....	3-2
<b>Use Parallel Computing Toolbox in Deployed Applications</b> .....	3-4
Export Cluster Profile .....	3-4
Link to Parallel Computing Toolbox Profile Within Your Code .....	3-4
Pass Parallel Computing Toolbox Profile at Run Time .....	3-5
Switch Between Cluster Profiles in Deployed Applications .....	3-5
Sample C Code to Load Cluster Profile .....	3-5
<b>Deploy Applications on Network Drives</b> .....	3-7
<b>MATLAB Compiler SDK Deployment Messages</b> .....	3-8

## Distributing Code to an End User

### 4

<b>MATLAB Runtime Component Cache and Deployable Archive Embedding</b> .....	4-2
---	-----

## Compiler Commands

### 5

<b>Compiler Tips</b> .....	5-2
Deploying Applications That Call the Java Native Libraries .....	5-2
Using the VER Function in a Compiled MATLAB Application .....	5-2

## Troubleshooting

### 6

<b>Common Issues</b> .....	6-2
<b>Compilation Failures</b> .....	6-3
<b>Testing Failures</b> .....	6-5
<b>Deployment Failures</b> .....	6-8
<b>Troubleshoot mbuild</b> .....	6-10
<b>Deployed Applications</b> .....	6-11

7

<b>MATLAB Runtime Path Settings for Development and Testing</b> .....	7-2
Path for Java Development on All Platforms .....	7-2
Path Modifications Required for Accessibility .....	7-2
Windows Settings for Development and Testing .....	7-2
Linux Settings for Development and Testing .....	7-2
OS X Settings for Development and Testing .....	7-2
<b>Set MATLAB Runtime Path for Deployment</b> .....	7-4
Library Path Environment Variables and MATLAB Runtime Folders .....	7-4
Windows .....	7-5
Linux .....	7-5
macOS .....	7-6
Set Path Permanently on UNIX .....	7-6
<b>MATLAB Compiler SDK Licensing</b> .....	7-8
Use MATLAB Compiler SDK Licenses for Development .....	7-8
<b>Deployment Product Terms</b> .....	7-9

Functions

8

C++ Utility Library Reference

A

<b>Data Conversion Restrictions for the C++ mxArray API</b> .....	A-2
<b>Primitive Types</b> .....	A-3
<b>C++ Utility Classes</b> .....	A-4

C++ MATLAB Data API

9

Workflow: C++ Shared Library using MATLAB Data API

10

<b>Integrate C++ Shared Libraries with MATLAB Data API</b> .....	10-2
Workflow to Create C++ Shared Library using Generic Interface .....	10-2

## Strongly Typed Cross Language Interface

# 11

<b>C++ MATLAB Data API Shared Library Support for Strongly Typed MATLAB Code</b> .....	<b>11-2</b>
<b>Create C++ MATLAB Data API Shared Library Header from Strongly Typed MATLAB Function</b> .....	<b>11-5</b>
Prerequisites .....	<b>11-5</b>
Create Function in MATLAB .....	<b>11-5</b>
Generate C++ Shared Library Header Using the mcc Command .....	<b>11-5</b>
Integrate C++ MATLAB Data API Shared Library Header with C++ Application .....	<b>11-6</b>
<b>Create C++ MATLAB Data API Shared Library Header from Strongly Typed MATLAB Classes Contained in Package</b> .....	<b>11-8</b>
Prerequisites .....	<b>11-8</b>
Files .....	<b>11-8</b>
Create Classes and Functions in MATLAB .....	<b>11-8</b>
Generate C++ Shared Library Header Using the mcc Command .....	<b>11-10</b>
Integrate C++ MATLAB Data API Shared Library Header with C++ Application .....	<b>11-12</b>
<b>Data Type Mappings Between C++ and Strongly Typed MATLAB Code</b> .....	<b>11-13</b>
C++ to MATLAB .....	<b>11-13</b>
MATLAB to C++ .....	<b>11-16</b>

# Installation and Configuration

---

- “Configure the mbuild Options File” on page 1-2
- “Solve Installation Problems” on page 1-3

## Configure the mbuild Options File

The `mbuild` utility compiles and links applications that integrate MATLAB generated shared libraries. Its options file specifies the compiler and linker settings used to build the application.

By default, the `mbuild` utility selects the appropriate compiler using preset default configuration.

To change the options used by the `mbuild` utility:

- 1** Use `mbuild -setup` to make a copy of the appropriate options file in your preferences folder.  
You can determine the path to the user preference folder using the MATLAB `prefdir` function.
- 2** Edit your copy of the options file to correspond to your specific needs, and save the modified file.



## Solve Installation Problems

You can contact MathWorks:

- Via the website at [www.mathworks.com](http://www.mathworks.com). On the MathWorks home page, click **My Account** to access your MathWorks Account, and follow the instructions.
- Via email at [service@mathworks.com](mailto:service@mathworks.com).



# Libraries

---

- “Call a C Shared Library” on page 2-2
- “Integrate C++ Shared Libraries with mxArray” on page 2-6
- “Use Multiple Shared Libraries in Single Application” on page 2-9
- “Understand the mclmcrtr Proxy Layer” on page 2-16
- “Call MATLAB Compiler SDK API Functions from C/C++” on page 2-17
- “Memory Management and Cleanup” on page 2-25
- “Write Applications for macOS” on page 2-26

## Call a C Shared Library

To use one or more MATLAB Compiler SDK generated C shared libraries in your C application:

- 1 Include the generated header file for each library in your application.  
 Each generated shared library has an associated header file named *libname.h*.
- 2 Initialize the MATLAB Runtime proxy layer by calling `mclmcrInitialize`.
- 3 Use `mclRunMain` to call the C function in your driver code that uses the MATLAB generated shared libraries.

`mclRunMain()` provides a convenient cross platform mechanism for wrapping the execution of MATLAB code in the shared library.

---

**Caution** Do not use `mclRunMain()` on Mac if your application brings up its own full graphical environment.

---

- 4 Declare variables and process input arguments.
- 5 Initialize the MATLAB Runtime by calling the `mclInitializeApplication` function. This function sets up the global MATLAB Runtime state and enables the construction of MATLAB Runtime instances.

Call the `mclInitializeApplication()` function once per application. It must be called before any other MATLAB API functions. You can pass application-level options to this function. `mclInitializeApplication()` returns a boolean status code.

---

**Caution** Avoid issuing `cd` commands from the driver application before calling `mclInitializeApplication`. Failure to do so can cause a failure in MATLAB Runtime initialization.

---

- 6 For each C shared library that you include in your application, call the initialization function for the library.

The initialization function performs library-local initialization. It unpacks the deployable archive and starts a MATLAB Runtime instance with the necessary information to execute the code in that archive. The library initialization function is named *libnameInitialize()*. This function returns a Boolean status code.

---

**Note** On Windows®, if you want to have your shared library call a MATLAB shared library, the MATLAB library initialization function (e.g., `<libname>Initialize`, `<libname>Terminate`, `mclInitialize`, `mclTerminate`) cannot be called from your shared library during the `DllMain(DLL_ATTACH_PROCESS)` call. This applies whether the intermediate shared library is implicitly or explicitly loaded. Place the call after `DllMain()`.

---

- 7 Invoke functions in the library, and process the results. (This is the main body of the program.)

---

**Note** If your driver application displays MATLAB figure windows, include a call to `mclWaitForFiguresToDie` before calling the `Terminate` functions and `mclTerminateApplication` in the following two steps.

---

- 8 When your application no longer needs a given library, call the termination function for the library.

The terminate function frees the resources associated with the library's MATLAB Runtime instance. The library termination function is named `libnameTerminate()`. Once a library has been terminated, the functions exported by the library cannot be called again in the application.

---

**Caution** Issuing a `<lib>Initialize` call after a `<lib>Terminate` call (whether or not the library is the same) causes unpredictable results.

---

- 9 When your application no longer needs to call any shared libraries, call the `mclTerminateApplication` API function.

This function frees application-level resources used by the MATLAB Runtime. Once you call this function, no further calls can be made to shared libraries in the application.

- 10 Clean up variables, close files, and exit.

The following example from `matrix.c` illustrates all of the above steps.

### Call a C Shared Library from Your C Driver Application

```

/*=====
 *
 * MATRIX.C Sample driver code that calls a shared library created
 *         using MATLAB Compiler SDK. Refer to the MATLAB Compiler
 *         SDK documentation for more information.
 *
 * Copyright 1984-2017 The MathWorks, Inc.
 *
 *=====*/

#include <stdio.h>

/* Include the MATLAB Runtime header file and the library specific header file
 * as generated by MATLAB Compiler SDK. */
#include "libmatrix.h"

/* This function is used to display a double matrix stored in an mxArray */
void display(const mxArray* in);

int run_main(int argc, const char **argv)
{
    mxArray *in1, *in2; /* Define input parameters */
    mxArray *out = NULL; /* and output parameters to be passed to the library functions */

    double data[] = {1,2,3,4,5,6,7,8,9};

    /* Create the input data */
    in1 = mxCreateDoubleMatrix(3,3,mxREAL);
    in2 = mxCreateDoubleMatrix(3,3,mxREAL);
    memcpy(mxGetPr(in1), data, 9*sizeof(double));
    memcpy(mxGetPr(in2), data, 9*sizeof(double));

    /* Call the library initialization routine and make sure that the
     * library was initialized properly. */
    if (!libmatrixInitialize()){
        fprintf(stderr,"Could not initialize the library.\n");
        return -2;
    }
    else

```

```
{
    /* Call the library function */
    mlfAddmatrix(1, &out, in1, in2);
    /* Display the return value of the library function */
    printf("The sum of the matrix with itself is:\n");
    display(out);
    /* Destroy the return value since this variable will be reused in
     * the next function call. Since we are going to reuse the variable,
     * we must set it to NULL. Refer to MATLAB Compiler SDK documentation
     * for more information. */
    mxDestroyArray(out);
    out=0;
    mlfMultiplymatrix(1, &out, in1, in2);
    printf("The product of the matrix with itself is:\n");
    display(out);
    mxDestroyArray(out);
    out=0;
    mlfEigmatrix(1, &out, in1);
    printf("The eigenvalues of the original matrix are:\n");
    display(out);
    mxDestroyArray(out);
    out=0;

    /* Call the library termination routine */
    libmatrixTerminate();

    /* Free the memory created */
    mxDestroyArray(in1);
    in1=0;
    mxDestroyArray(in2);
    in2=0;
}

/* Note that you should call mclTerminateApplication at the end of
 * your application.
 */
mclTerminateApplication();
return 0;
}

/*DISPLAY This function will display the double matrix stored in an mxArray.
 * This function assumes that the mxArray passed as input contains double
 * array.
 */
void display(const mxArray* in)
{
    size_t i=0, j=0; /* loop index variables */
    size_t r=0, c=0; /* variables to store the row and column length of the matrix */
    double *data; /* variable to point to the double data stored within the mxArray */

    /* Get the size of the matrix */
    r = mxGetM(in);
    c = mxGetN(in);
    /* Get a pointer to the double data in mxArray */
    data = mxGetPr(in);

    /* Loop through the data and display it in matrix format */
```

```

    for( i = 0; i < c; i++ )
    {
        for( j = 0; j < r; j++ )
        {
            printf("%4.2f\t",data[j*c+i]);
        }
        printf("\n");
    }
    printf("\n");
}

int main(int argc, const char ** argv)
{
    /* Call the mclInitializeApplication routine. Make sure that the application
    * was initialized properly by checking the return status. This initialization
    * has to be done before calling any MATLAB APIs or MATLAB Compiler SDK
    * generated shared library functions. */
    if( !mclInitializeApplication(NULL,0) )
    {
        fprintf(stderr, "Could not initialize the application.\n");
        return -1;
    }
    return mclRunMain((mclMainFcnType)run_main, argc, argv);
}

```

## Restrictions When Using MATLAB Function loadlibrary

You cannot use the MATLAB function `loadlibrary` in MATLAB to load a C shared library built with MATLAB Compiler SDK.

For more information about using `loadlibrary`, see “Calling Shared Libraries in Deployed Applications”.

## See Also

`mclmcrInitialize` | `mclRunMain` | `mclInitializeApplication` | `mclTerminateApplication`  
| `mclWaitForFiguresToDie`

## More About

- “Call MATLAB Compiler SDK API Functions from C/C++” on page 2-17
- “Understand the mclmcr Proxy Layer” on page 2-16
- “Create a C Shared Library with MATLAB Code”
- “Create C/C++ Shared Libraries from Command Line”

## Integrate C++ Shared Libraries with mxArray

### C++ Shared Library Wrapper

When you create a C++ shared library using MATLAB Compiler SDK, the compiler generates a wrapper file and a header file. The header file contains all of the entry points for the compiled MATLAB functions.

### C++ libmatrix Example

The examples on this page reference the C++ shared library `libmatrix`. To create this library, complete the example “Generate a C++ mxArray API Shared Library and Build a C++ Application”.

Ensure both files are in the current folder, then compile the `matrix_mwarray.cpp` application using `mbuild`.

```
mbuild matrix_mwarray.cpp libmatrix.lib
```

---

**Note** Due to name mangling in C++, you must compile your C++ application with the same version of the third-party compiler that you use to compile your C++ shared library.

---

### Arrays in C++ Applications

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

---

**Caution** Avoid issuing `cd` commands from the C++ application prior to calling `mclInitializeApplication`. Failure to do so can cause a failure in MATLAB Runtime initialization.

---

For information about how MATLAB Compiler SDK uses a proxy layer for the linked libraries, see “Understand the `mclmcr` Proxy Layer” on page 2-16.

### Incorporate C++ Shared Library into Application

There are two main differences to note when using a C++ shared library:

- Interface functions use the `mwArray` type to pass arguments, rather than the `mxArray` type used with C shared libraries.
- C++ exceptions are used to report errors to the caller. Therefore, all calls must be wrapped in a `try-catch` block.



## Exported Function Signature

The C++ shared library target generates two sets of interfaces for each MATLAB function. For more information, see “Functions Generated from MATLAB Files” on page 2-20. The generic signature of the exported C++ functions is as follows:

### MATLAB Functions with No Return Values

```
bool MW_CALL_CONV <function-name>(<const_mwArray_references>);
```

### MATLAB Functions with at Least One Return Value

```
bool MW_CALL_CONV <function-name>(int <number_of_return_values>,
    <mwArray_references>, <const_mwArray_references>);
```

In this case, *const\_mwArray\_references* represents a comma-separated list of references of type `const mxArray&` and *mwArray\_references* represents a comma-separated list of references of type `mxArray&`. For example, in the `libmatrix` library, the C++ interface to the `addmatrix` MATLAB function is generated as:

```
void addmatrix(int nargout, mxArray& a, const mxArray& a1,
    const mxArray& a2);
```

where `a` is an output parameter and `a1` and `a2` are input parameters.

Input arguments passed to the MATLAB function via `varargin` must be passed via a single `mxArray` that is a cell array. Each element in the cell array must constitute an input argument. Output arguments retrieved from the MATLAB function via `varargout` must be retrieved via a single `mxArray` that is a cell array. Each element in the cell array will constitute an output argument. The number of elements in the cell array will be equal to `number_of_return_values` - the number of named output parameters. Also note that,

- If the MATLAB function takes a `varargin` argument, the C++ function must be passed an `mxArray` corresponding to that `varargin`, even if the `mxArray` is empty.
- If the MATLAB function takes a `varargout` argument, the C++ function must be passed an `mxArray` corresponding to that `varargout`, even if `number_of_return_values` is set to the number of named output arguments, which means meaning that `varargout` will be empty.
- The `varargout` argument needs to follow any named output arguments and precede any input arguments.
- The `varargin` argument needs to be the last argument.

## Error Handling

C++ interface functions handle errors during execution by throwing a C++ exception. Use the `mwException` class for this purpose. Your application can catch `mwExceptions` and query the `what()` method to get the error message. To correctly handle errors when calling the C++ interface functions, wrap each call inside a `try-catch` block.

```
    try
    {
        ...
        (call function)
        ...
    }
    catch (const mwException& e)
    {
```

```
        ...  
        (handle error)  
        ...  
    }
```

The `matrix_mwarray.cpp` application illustrates the typical way to handle errors when calling the C++ interface functions.

### **Working with C++ Shared Libraries and Sparse Arrays**

The MATLAB Compiler SDK C/C++ API includes static factory methods for working with sparse arrays.

For a complete list of the methods, see “C++ Utility Classes” on page A-4.

### **See Also**

#### **More About**

- “Generate a C++ mxArray API Shared Library and Build a C++ Application”
- “Call MATLAB Compiler SDK API Functions from C/C++” on page 2-17
- “C++ Utility Classes” on page A-4
- “Use Multiple Shared Libraries in Single Application” on page 2-9

## Use Multiple Shared Libraries in Single Application

### In this section...

“Initialize and Terminate Multiple Shared Libraries” on page 2-9

“Work with MATLAB Function Handles” on page 2-10

“Work with Objects” on page 2-13

When developing applications that use multiple MATLAB shared libraries, consider the following:

- Each MATLAB shared library must be initialized separately.
- Each MATLAB shared library must be terminated separately.
- MATLAB function handles cannot be shared between shared libraries.
- MATLAB figure handles cannot be shared between shared libraries.
- MATLAB objects cannot be shared between shared libraries.
- C, Java®, and .NET objects cannot be shared between shared libraries.
- Executable data stored in cell arrays and structures cannot be shared between shared libraries

### Initialize and Terminate Multiple Shared Libraries

To initialize and terminate multiple shared libraries:

- 1 Initialize the MATLAB Runtime using `mclmcrInitialize()`.
- 2 Call the portion of the application that executes the MATLAB code using `mclRunMain()`.
- 3 Before initializing the shared libraries, initialize the MATLAB application state using `mclInitializeApplication()`.
- 4 For each MATLAB shared library, call the generated initialization function, `libraryInitialize()`.
- 5 Add the code for working with the MATLAB code.
- 6 For each MATLAB shared library, release the resources used by the library using the generated termination function, `libraryTerminate()`.
- 7 Release the resources used by the MATLAB Runtime by calling `mclTerminateApplication()`.

This example shows the use of two shared libraries.

#### Example driver code

```
#include <stdio.h>
#include "libAddMatrix.h"
#include "libSubMatrix.h"

int run_main(int argc, const char *argv[])
{
    if( !mclInitializeApplication(NULL,0) )
    {
        fprintf(stderr, "Could not initialize the application.\n");
        return -1;
    }

    if (!libAddMatrixInitialize())
    {
```

```

        fprintf(stderr,"Could not initialize the AddMatrix library.\n");
        return -2;
    }

    if (!libSubMatrixInitialize())
    {
        fprintf(stderr,"Could not initialize the SubMatrix library.\n");
        return -2;
    }

    try
    {
        ...
    }
    catch (const mwException& e)
    {
        std::cerr << e.what() << std::endl;
        return -2;
    }
    catch (...)
    {
        std::cerr << "Unexpected error thrown" << std::endl;
        return -3;
    }

    libAddMatrixTerminate();

    libSubMatrixTerminate();

    mclTerminateApplication();
    return 0;
}

int main(int ac, const char *av[])
{
    int err = 0;
    mclmcrInitialize();
    err = mclRunMain((mclMainFcnType) run_main, ac, av);
    return err;
}

```

## Work with MATLAB Function Handles

A MATLAB function handle can be passed back and forth between a MATLAB Runtime instance and an application. However, it cannot be passed from one MATLAB Runtime instance to another. For example, suppose that you had two MATLAB functions, `get_plot_handle` and `plot_xy`, and `plot_xy` used the function handle created by `get_plot_handle`.

```

% Saved as get_plot_handle.m
function h = get_plot_handle(lnSpec, lnWidth, mkEdge, mkFace, mkSize)
h = @draw_plot;
function draw_plot(x, y)
    plot(x, y, lnSpec, ...
        'LineWidth', lnWidth, ...
        'MarkerEdgeColor', mkEdge, ...
        'MarkerFaceColor', mkFace, ...
        'MarkerSize', mkSize)

```

```

    end
end

% Saved as plot_xy.m
function plot_xy(x, y, h)
h(x, y);
end

```

If you packaged them into two separate shared libraries, the call to `plot_xy` would throw an exception.

### Example driver code

```

#include <stdio.h>
#include "get_plot_handle.h"
#include "plot_xy.h"

int run_main(int argc, const char *argv[])
{
    if( !mclInitializeApplication(NULL,0) )
    {
        fprintf(stderr, "Could not initialize the application.\n");
        return -1;
    }

    if (!get_plot_handleInitialize())
    {
        fprintf(stderr,
            "Could not initialize the get_plot_handle library.\n");
        return -2;
    }

    if (!plot_xyInitialize())
    {
        fprintf(stderr,"Could not initialize the plot_xy library.\n");
        return -2;
    }

    try
    {
        mxArray lnSpec('--rs');
        mxArray lnWidth;
        lnWidth = 2.0;
        mxArray mkEdge('k');
        mxArray mkFace('g');
        mxArray mkSize;
        mkSize = 10.0;
        mxArray plot;
        get_plot_handle(1, plot, lnSpec, lnWidth, mkEdge, mkFace, mkSize);

        double x_data[] = {1,2,3,4,5,6,7,8,9};
        double y_data[] = {2,6,12,20,30,42,56,72,90};
        mxArray x(9, 1, mxDOUBLE_CLASS, mxREAL);
        mxArray y(9, 1, mxDOUBLE_CLASS, mxREAL);
        x.SetData(x_data, 9);
        y.SetData(y_data, 9);
        plot_xy(x, y, plot);
    }
}

```

```
    }
    catch (const mwException& e)
    {
        std::cerr << e.what() << std::endl;
        return -2;
    }
    catch (...)
    {
        std::cerr << "Unexpected error thrown" << std::endl;
        return -3;
    }

    get_plot_handleTerminate();

    plot_xyTerminate();

    mclTerminateApplication();
    return 0;
}

int main(int ac, const char *av[])
{
    int err = 0;
    mclmcrInitialize();
    err = mclRunMain((mclMainFcnType) run_main, ac, av);
    return err;
}
```

One way to handle the situation is to package both functions into a single shared library. For example, if you called the shared library `plot_functions`, your application would only need one call to initialize the function and you could pass the function handle for `plot_xy` without error.

### Example driver code

```
#include <stdio.h>
#include "plot_functions.h"

int run_main(int argc, const char *argv[])
{
    if( !mclInitializeApplication(NULL,0) )
    {
        fprintf(stderr, "Could not initialize the application.\n");
        return -1;
    }

    if (plot_functionsInitialize())
    {
        fprintf(stderr,
            "Could not initialize the plot_functions library.\n");
        return -2;
    }

    try
    {
        mwArray lnSpec('--rs');
        mwArray lnWidth;
        lnWidth = 2.0;
    }
```

```

    mxArray mkEdge('k');
    mxArray mkFace('g');
    mxArray mkSize;
    mkSize = 10.0;
    mxArray plot;
    get_plot_handle(1, plot, lnSpec, lnWidth, mkEdge, mkFace, mkSize);

    double x_data[] = {1,2,3,4,5,6,7,8,9};
    double y_data[] = {2,6,12,20,30,42,56,72,90};
    mxArray x(9, 1, mxDOUBLE_CLASS, mxREAL);
    mxArray y(9, 1, mxDOUBLE_CLASS, mxREAL);
    x.SetData(x_data, 9);
    y.SetData(y_data, 9);
    plot_xy(x, y, plot);
}
catch (const mxArrayException& e)
{
    std::cerr << e.what() << std::endl;
    return -2;
}
catch (...)
{
    std::cerr << "Unexpected error thrown" << std::endl;
    return -3;
}

plot_functionsTerminate();

mclTerminateApplication();
return 0;
}

int main(int ac, const char *av[])
{
    int err = 0;
    mclmcrInitialize();
    err = mclRunMain((mclMainFcnType) run_main, ac, av);
    return err;
}

```

## Work with Objects

MATLAB Compiler SDK enables you to return the following types of objects from the MATLAB Runtime to your application code:

- MATLAB
- C++
- .NET
- Java
- Python®

However, you cannot pass an object created in one MATLAB Runtime instance into a different MATLAB Runtime instance. This conflict can happen when a function that returns an object and a function that manipulates that object are packaged into different shared libraries.

For example, say that you develop two functions. The first creates a bank account for a customer. The second transfers funds between two accounts.

```
% Saved as account.m
classdef account < handle

    properties
        name
    end

    properties (SetAccess = protected)
        balance = 0
        number
    end

    methods
        function obj = account(name)
            obj.name = name;
            obj.number = round(rand * 1000);
        end

        function deposit(obj, deposit)
            new_bal = obj.balance + deposit;
            obj.balance = new_bal;
        end

        function withdraw(obj, withdrawl)
            new_bal = obj.balance - withdrawl;
            obj.balance = new_bal;
        end
    end
end

% Saved as open_acct .m
function acct = open_acct(name, open_bal )

    acct = account(name);

    if open_bal > 0
        acct.deposit(open_bal);
    end

end

% Saved as transfer.m
function transfer(source, dest, amount)

    if (source.balance > amount)
        dest.deposit(amount);
        source.withdraw(amount);
    end

end
```

If you packaged `open_acct.m` and `transfer.m` into separate shared libraries, you could not transfer funds using accounts created with `open_acct`. The call to `transfer` would throw an



exception. One way of resolving this is to package both functions into a single shared library. You could also refactor the application so as not to pass MATLAB objects to the functions.

## **See Also**

### **More About**

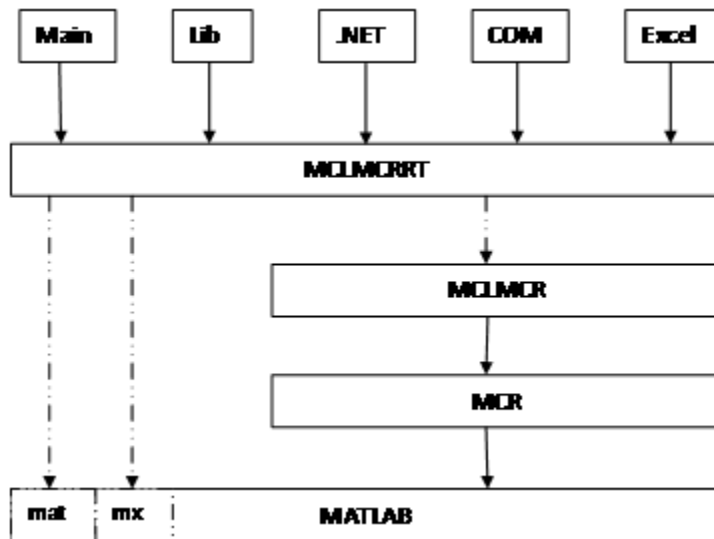
- “Call a C Shared Library” on page 2-2

## Understand the mclmcr rt Proxy Layer

All application and software components generated by MATLAB Compiler™ and MATLAB Compiler SDK need to link against only one MATLAB library, `mclmcr rt`. This library provides a proxy API for all the public functions in MATLAB libraries used for matrix operations, MAT-file access, utility and memory management, and application MATLAB Runtime. The `mclmcr rt` library lies between deployed MATLAB code and these other version-dependent libraries, providing the following functionality:

- Ensures that multiple versions of the MATLAB Runtime can coexist
- Provides a layer of indirection
- Ensures applications are thread-safe
- Loads the dependent (re-exported) libraries dynamically

The relationship between `mclmcr rt` and other MATLAB libraries is shown in the following figure.



### The MCLMCRRT Proxy Layer

In the figure, solid arrows designate static linking and dotted arrows designate dynamic linking. The figure illustrates how the `mclmcr rt` library layer sits above the `mclmcr` and `mcr` libraries. The `mclmcr` library contains the run-time functionality of the deployed MATLAB code. The `mcr` module ensures each bundle of deployed MATLAB code runs in its own context at run time. The `mclmcr rt` proxy layer, in addition to loading the `mclmcr`, also dynamically loads the `MX` and `MAT` modules, primarily for `mxArray` manipulation. For more information, see the MathWorks® Support database and search for information on the MSVC shared library.

**Caution** Deployed applications must only link to the `mclmcr rt` proxy layer library (`mclmcr rt.lib` on Windows, `mclmcr rt.so` on Linux®, and `mclmcr rt.dylib` on Macintosh). Do not link to the other libraries shown in the figure, such as `mclmcr`, `libmx`, and so on.

## Call MATLAB Compiler SDK API Functions from C/C++

### Shared Library Functions

A C or C++ shared library generated by MATLAB Compiler SDK contains at least seven functions. There are three generated functions to manage library initialization and termination, one each for printed output and error messages, and two generated functions for each MATLAB function included in the library.

To generate the functions described in this section, first copy the folder `matlabroot\extern\examples\compilersdk\c_cpp\triangle` to your current working directory.

Create a shared library named `libtriangle` that contains the function `sierpinski.m` by following the procedure in “Create a C Shared Library with MATLAB Code”, “Generate a C++ mxArray API Shared Library and Build a C++ Application”, or “Generate a C++ MATLAB Data API Shared Library and Build a C++ Application”, depending on the desired target application.

### Type of Application

Once your shared library is created, execute the `mbuild` command that corresponds to your target language. This command uses your C/C++ compiler to compile the code and link the driver code against the MATLAB Compiler SDK generated C/C++ shared library.

For a C application, use `mbuild triangle.c libtriangle.lib`.

For a C++ mxArray API application, use `mbuild triangle_mwarray.cpp libtriangle.lib`

For a C++ MATLAB Data API application, use `mbuild triangle_mda.cpp`

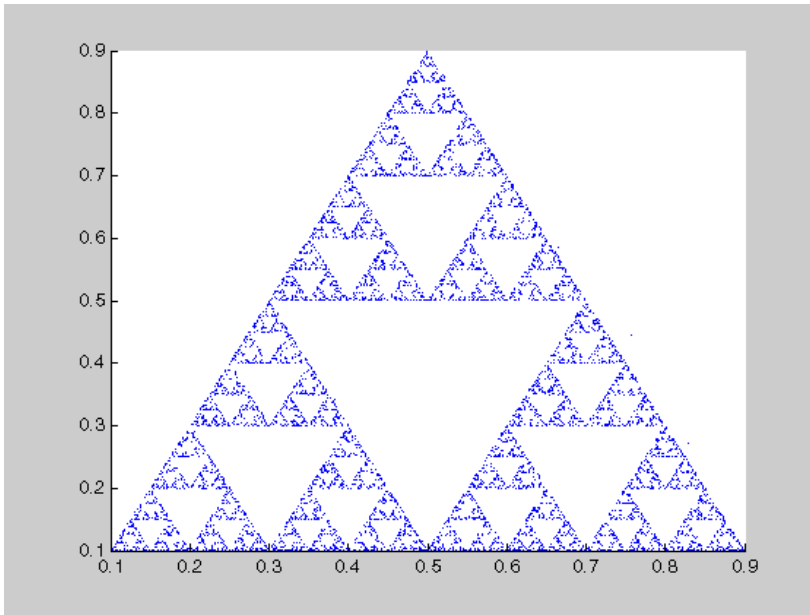
---

**Note** The `.lib` extension is for Windows. On Mac, the file extension is `.dylib`, and on UNIX® it is `.so`.

This command assumes that the C/C++ shared library, the driver code, and the corresponding header file are in the current working folder.

---

These commands create a main program named after the selected source code file that call function in the `libtriangle` shared library. The library exports a single function (contained in `sierpinski.m`) that uses a simple iterative algorithm to generate the fractal known as Sierpinski's Triangle. The main program can optionally take a single numeric argument which specifies the number of points used to generate the fractal. For example, `triangle 8000` generates a diagram with 8,000 points.



## Structure of Programs That Call Shared Libraries

All programs that call MATLAB Compiler SDK generated shared libraries have roughly the same structure:

- 1 Declare variables and process/validate input arguments.
- 2 Call `mclInitializeApplication` and test for success. This function sets up the global MATLAB Runtime state and enables the construction of MATLAB Runtime instances.
- 3 Call `<library>Initialize[WithHandlers]` once for each library to create the MATLAB Runtime instance required by the library.
- 4 Invoke functions in the library and process the results in the main body of the program.
- 5 Call `<library>Terminate` once for each library to destroy the associated MATLAB Runtime.
- 6 Call `mclTerminateApplication` to free resources associated with the global MATLAB Runtime state.
- 7 Clean up variables, close files, etc., and exit.

## Library Initialization and Termination Functions

The library initialization and termination functions create and destroy, respectively, the MATLAB Runtime instance required by the shared library. You must call the initialization function before you invoke any of the other functions in the shared library, and you should call the termination function after you are finished making calls into the shared library, or you risk leaking memory.

There are two forms of the initialization function `<library>Initialize[WithHandlers]` and one of the termination function `<library>Terminate`. The name of your generated C/C++ shared library is used as part of the function name. The simpler of the two initialization functions takes no arguments; most likely, this is the version your application will call. In this example, this form of the initialization function is called `libtriangleInitialize`.

```
bool libtriangleInitialize(void)
```

This function creates a MATLAB Runtime instance using the default print and error handlers, and other information generated during the compilation process.

However, if you want more control over how printed output and error messages are handled, call the second form of the function, which takes two arguments. In this example, this form of the initialization function is called `libtriangleInitializeWithHandlers`.

```
bool libtriangleInitializeWithHandlers(
    mclOutputHandlerFcn error_handler,
    mclOutputHandlerFcn print_handler
)
```

By calling this function syntax, you can provide your own versions of the print and error handling routines called by the MATLAB Runtime. Each of these routines has the same signature (for complete details, see “Print and Error Handling Functions” on page 2-19). By overriding the defaults, you can control how output is displayed and, for example, whether or not it goes into a log file.

---

**Note** Before calling either form of the library initialization routine, you must first call `mclInitializeApplication` to set up the global MATLAB Runtime state. For more information, see “Call a C Shared Library” on page 2-2.

---

On Microsoft® Windows platforms, MATLAB Compiler SDK generates an additional initialization function: the standard Microsoft DLL initialization function `DllMain`.

```
BOOL WINAPI DllMain(HINSTANCE hInstance, DWORD dwReason,
    void *pv)
```

The generated `DllMain` performs a very important service; it locates the directory in which the shared library is stored on disk. This information is used to find the deployable archive, without which the application will not run. If you modify the generated `DllMain` (not recommended), make sure you preserve this part of its functionality.

Library termination is simple.

```
void libtriangleTerminate(void)
```

Call this function (once for each library) before calling `mclTerminateApplication`.

## Print and Error Handling Functions

By default, MATLAB Compiler SDK generated applications and shared libraries send printed output to standard output and error messages to standard error. MATLAB Compiler SDK generates a default print handler and a default error handler that implement this policy. If you'd like to change this behavior, you must write your own error and print handlers and pass them in to the appropriate generated initialization function.

You may replace either, both, or neither of these two functions. The MATLAB Runtime sends all regular output through the print handler and all error output through the error handler. Therefore, if you redefine either of these functions, the MATLAB Runtime will use your version of the function for all the output that falls into class for which it invokes that handler.

The default print handler takes the following form.

```
static int mclDefaultPrintHandler(const char *s)
```

The implementation is straightforward; it takes a string, prints it on standard output, and returns the number of characters printed. If you override or replace this function, your version must also take a string and return the number of characters “handled.” The MATLAB Runtime calls the print handler when an executing MATLAB file makes a request for printed output, e.g., via the MATLAB function `disp`. The print handler does not terminate the output with a carriage return or line feed.

The default error handler has the same form as the print handler.

```
static int mclDefaultErrorHandler(const char *s)
```

However, the default implementation of the print handler is slightly different. It sends the output to the standard error output stream, but if the string does not end with carriage return, the error handler adds one. If you replace the default error handler with one of your own, you should perform this check as well, or some of the error messages printed by the MATLAB Runtime will not be properly formatted.

For an example on using custom print and error handling functions in your application, see the files located in `matlabroot\extern\examples\compilersdk\c_cpp\catcherror`.

---

**Caution** The error handler does not handle the actual errors, but rather the message produced after the errors have been caught and handled inside the MATLAB Runtime. You cannot use this function to modify the error handling behavior of the MATLAB Runtime -- use the `try` and `catch` statements in your MATLAB files if you want to control how a MATLAB Compiler SDK generated application responds to an error condition.

---

---

**Note** If you provide alternate C++ implementations of either `mclDefaultPrintHandler` or `mclDefaultErrorHandler`, then functions must be declared `extern "C"`. For example:

```
extern "C" int myPrintHandler(const char *s);
```

---

## Functions Generated from MATLAB Files

For each MATLAB file specified on the MATLAB Compiler SDK command line, the product generates two functions, the `mlx` function and the `mlf` function. Each of these generated functions performs the same action (calls your MATLAB file function). The two functions have different names and present different interfaces. The name of each function is based on the name of the first function in the MATLAB file (`sierpinski`, in this example); each function begins with a different three-letter prefix.

---

**Note** For C shared libraries, MATLAB Compiler SDK generates the `mlx` and `mlf` functions as described in this section. For C++ shared libraries, the product generates the `mlx` function the same way it does for the C shared library. However, the product generates a modified `mlf` function with these differences:

- The `mlf` before the function name is dropped to keep compatibility with R13.
  - The arguments to the function are `mwArray` instead of `mxAArray`.
-

### mlx Interface Function

The function that begins with the prefix `mlx` takes the same type and number of arguments as a MATLAB MEX-function. (See the External Interfaces documentation for more details on MEX-functions.) The first argument, `nlhs`, is the number of output arguments, and the second argument, `plhs`, is a pointer to an array that the function will fill with the requested number of return values. (The “lhs” in these argument names is short for “left-hand side” -- the output variables in a MATLAB expression are those on the left-hand side of the assignment operator.) The third and fourth parameters are the number of inputs and an array containing the input variables.

```
void mlxSierpinski(int nlhs, mxArray *plhs[], int nrhs,
                  mxArray *prhs[])
```

### mlf Interface Function

The second of the generated functions begins with the prefix `mlf`. This function expects its input and output arguments to be passed in as individual variables rather than packed into arrays. If the function is capable of producing one or more outputs, the first argument is the number of outputs requested by the caller.

```
void mlfSierpinski(int nargout, mxArray** x, mxArray** y,
                  mxArray* iterations, mxArray* draw)
```

In both cases, the generated functions allocate memory for their return values. If you do not delete this memory (via `mxDestroyArray`) when you are done with the output variables, your program will leak memory.

Your program may call whichever of these functions is more convenient, as they both invoke your MATLAB file function in an identical fashion. Most programs will likely call the `mlf` form of the function to avoid managing the extra arrays required by the `mlx` form. The example program `triangle.c` calls `mlfSierpinski`.

```
mlfSierpinski(2, &x, &y, iterations, draw);
```

In this call, the caller requests two output arguments, `x` and `y`, and provides two inputs, `iterations` and `draw`.

If the output variables you pass in to an `mlf` function are not `NULL`, the `mlf` function will attempt to free them using `mxDestroyArray`. This means that you can reuse output variables in consecutive calls to `mlf` functions without worrying about memory leaks. It also implies that you must pass either `NULL` or a valid MATLAB array for all output variables or your program will fail because the memory manager cannot distinguish between a non-initialized (invalid) array pointer and a valid array. It will try to free a pointer that is not `NULL` -- freeing an invalid pointer usually causes a segmentation fault or similar fatal error.

### Using varargin and varargout in a MATLAB Function Interface

If your MATLAB function interface uses `varargin` or `varargout`, you must pass them as cell arrays. For example, if you have `N` `varargins`, you need to create one cell array of size 1-by-`N`. Similarly, `varargouts` are returned back as one cell array. The length of the `varargout` is equal to the number of return values specified in the function call minus the number of actual variables passed. As in the MATLAB software, the cell array representing `varargout` has to be the last return variable (the variable preceding the first input variable) and the cell array representing `varargins` has to be the last formal parameter to the function call.

For information on creating cell arrays, refer to the C MEX function interface in the External Interfaces documentation.

For example, consider this MATLAB file interface:

```
[a,b,varargout] = myfun(x,y,z,varargin)
```

The corresponding C interface for this is

```
void mlfMyfun(int numOfRetVars, mxArray **a, mxArray **b,
             mxArray **varargout, mxArray *x, mxArray *y,
             mxArray *z, mxArray *varargin)
```

In this example, the number of elements in `varargout` is  $(\text{numOfRetVars} - 2)$ , where 2 represents the two variables, `a` and `b`, being returned. Both `varargin` and `varargout` are single row, multiple column cell arrays.

---

**Caution** The C++ shared library interface does not support `varargin` with zero (0) input arguments. Calling your program using an empty `mwArray` results in the packaged library receiving an empty array with `nargin = 1`. The C shared library interface allows you to call `mlfF00(NULL)` (the packaged MATLAB code interprets this as `nargin=0`). However, calling `F00((mwArray)NULL)` with the C++ shared library interface causes the packaged MATLAB code to see an empty array as the first input and interprets `nargin=1`.

For example, package some MATLAB code as a C++ shared library using `varargin` as the MATLAB function's list of input arguments. Have the MATLAB code display the variable `nargin`. Call the library with function `F00()` and it won't package, producing this error message:

```
... 'F00' : function does not take 0 arguments
```

Call the library as:

```
mwArray junk;
F00(junk);
```

or

```
F00((mwArray)NULL);
```

At runtime, `nargin=1`. In MATLAB, `F00()` is `nargin=0` and `F00([])` is `nargin=1`.

---

### C++ Interfaces for MATLAB Functions Using `varargin` and `varargout`

The C++ `mlx` interface for MATLAB functions does not change even if the functions use `varargin` or `varargout`. However, the C++ function interface (the second set of functions) changes if the MATLAB function is using `varargin` or `varargout`.

For examples, view the generated code for various MATLAB function signatures that use `varargin` or `varargout`.

---

**Note** For simplicity, only the relevant part of the generated C++ function signature is shown in the following examples.

---



**function varargout = foo(varargin)**

For this MATLAB function, the following C++ overloaded functions are generated:

No input no output:  
 void foo()

Only inputs:  
 void foo(const mxArray& varargin)

Only outputs:  
 void foo(int nargout, mxArray& varargin)

Most generic form that has both inputs and outputs:  
 void foo(int nargout, mxArray& varargin,  
           const mxArray& varargin)

**function varargout = foo(i1, i2, varargin)**

For this MATLAB function, the following C++ overloaded functions are generated:

Most generic form that has outputs and all the inputs  
 void foo(int nargout, mxArray& varargin, const  
           mxArray& i1, const  
           mxArray& i2, const  
           mxArray& varargin)

Only inputs:  
 void foo(const mxArray& i1,  
           const mxArray& i2, const mxArray& varargin)

**function [o1, o2, varargin] = foo(varargin)**

For this MATLAB function, the following C++ overloaded functions are generated:

Most generic form that has all the outputs and inputs  
 void foo(int nargout, mxArray& o1, mxArray& o2,  
           mxArray& varargin,  
           const mxArray& varargin)

Only outputs:  
 void foo(int nargout, mxArray& o1, mxArray& o2,  
           mxArray& varargin)

**function [o1, o2, varargin] = foo(i1, i2, varargin)**

For this MATLAB function, the following C++ overloaded function is generated:

Most generic form that has all the outputs and  
   all the inputs  
 void foo(int nargout, mxArray& o1, mxArray& o2,  
 mxArray& varargin,  
           const mxArray& i1, const mxArray& i2,  
           const mxArray& varargin)

## **Retrieving MATLAB Runtime State Information While Using Shared Libraries**

When using shared libraries, you may call functions to retrieve specific information from the MATLAB Runtime state. For details, see “Set and Retrieve MATLAB Runtime Data for Shared Libraries”.

### **See Also**

`mbuild`

### **More About**

- “Create a C Shared Library with MATLAB Code”
- “Generate a C++ mxArray API Shared Library and Build a C++ Application”
- “Generate a C++ MATLAB Data API Shared Library and Build a C++ Application”
- “Call a C Shared Library” on page 2-2

# Memory Management and Cleanup

In this section...
“Overview” on page 2-25
“Passing mxArray to Shared Libraries” on page 2-25

## Overview

Generated C++ code provides consistent garbage collection via the object destructors and the MATLAB Runtime's internal memory manager optimizes to avoid heap fragmentation.

If memory constraints are still present on your system, try preallocating arrays in MATLAB. This will reduce the number of calls to the memory manager, and the degree to which the heap fragments.

## Passing mxArray to Shared Libraries

When an mxArray is created in an application which uses the MATLAB Runtime, it is created in the managed memory space of the MATLAB Runtime.

Therefore, it is very important that you never create mxArray (or call any other MATLAB function) before calling `mclInitializeApplication`.

It is safe to call `mxDestroyArray` when you no longer need a particular mxArray in your code, even when the input has been assigned to a persistent or global variable in MATLAB. MATLAB uses reference counting to ensure that when `mxDestroyArray` is called, if another reference to the underlying data still exists, the memory will not be freed. Even if the underlying memory is not freed, the mxArray passed to `mxDestroyArray` will no longer be valid.

For more information about `mclInitializeApplication` and `mclTerminateApplication`, see “Call a C Shared Library” on page 2-2.

For more information about mxArray, see “C Matrix API”.

## Write Applications for macOS

### In this section...

“Objective-C/C++ Applications for Apple’s Cocoa API” on page 2-26

“Where’s the Example Code?” on page 2-26

“Preparing Your Apple Xcode Development Environment” on page 2-26

“Build and Run the Sierpinski Application” on page 2-27

“Running the Sierpinski Application” on page 2-28

### Objective-C/C++ Applications for Apple’s Cocoa API

Apple Xcode, implemented in the Objective-C language, is used to develop applications using the Cocoa framework, the native object-oriented API for the Mac OS operating system.

This article details how to create a graphical MATLAB application with Objective C and Cocoa, and then deploy it using MATLAB Compiler SDK.

### Where’s the Example Code?

You can find example Apple Xcode, header, and project files in *matlabroot/extern/examples/compilersdk/c\_cpp/triangle/xcode*.

### Preparing Your Apple Xcode Development Environment

To run this example, you should have prior experience with the Apple Xcode development environment and the Cocoa framework.

The example in this article is ready to build and run on page 2-27. However, before you build and run your own applications, you must do the following (as has been done in our example code on page 2-26):

- 1 Build the shared library with MATLAB Compiler SDK using either Library Compiler, `compiler.build.cppSharedLibrary`, or `mcc`.
- 2 Compile application code against the library’s header file and link the application against the component library and `libmwmclmcr`.
- 3 In your Apple Xcode project:
  - Specify `mcc` in the project target (Build Component Library in the example code on page 2-26).
  - Specify target settings in `HEADER_SEARCH_PATHS`.
    - Specify directories containing the library header.
    - Specify the path *matlabroot/extern/include*.
    - Define `MWINSTALL_ROOT`, which establishes the install route using a relative path.
  - Set `LIBRARY_SEARCH_PATHS` to any directories containing the shared library, as well as to the path *matlabroot/runtime/maci64*.

## Build and Run the Sierpinski Application

In this example, deploy the graphical Sierpinski function `sierpinski.m`, located at `matlabroot/extern/examples/compiler/sdk/c_cpp/triangle`.

```
function [x, y] = sierpinski(iterations, draw)
% SIERPINSKI Calculate (optionally draw) the points
% in Sierpinski's triangle

% Copyright 2004 The MathWorks, Inc.

% Three points defining a nice wide triangle
points = [0.5 0.9 ; 0.1 0.1 ; 0.9 0.1];

% Select an initial point
current = rand(1, 2);

% Create a figure window
if (draw == true)
    f = figure;
    hold on;
end

% Pre-allocate space for the results, to improve performance
x = zeros(1,iterations);
y = zeros(1,iterations);

% Iterate
for i = 1:iterations

    % Select point at random
    index = floor(rand * 3) + 1;

    % Calculate midpoint between current point and random point
    current(1) = (current(1) + points(index, 1)) / 2;
    current(2) = (current(2) + points(index, 2)) / 2;

    % Plot that point
    if draw, line(current(1),current(2));, end
    x(i) = current(1);
    y(i) = current(2);

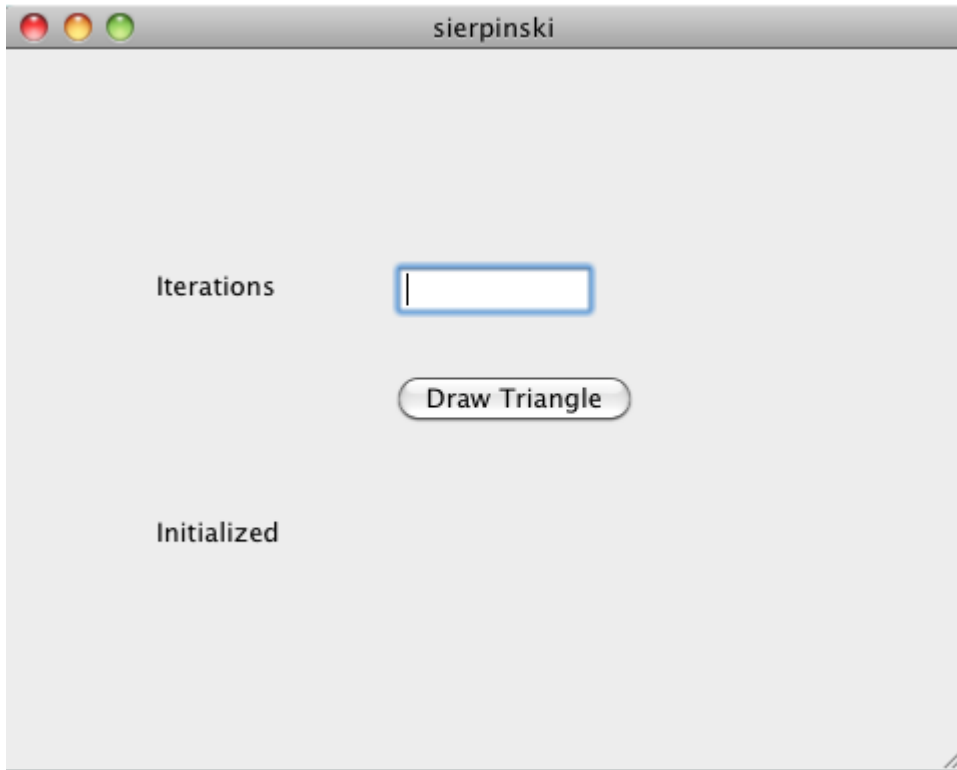
end

if (draw)
    drawnow;
end
```

- 1 Using the Mac Finder, locate the Apple Xcode project (`matlabroot/extern/examples/compiler/sdk/c_cpp/triangle/xcode`). Copy files to a working directory to run this example, if needed.
- 2 Open `sierpinski.xcodeproj`. The development environment starts.
- 3 In the **Groups and Files** pane, select **Targets**.
- 4 Click **Build and Run**. The make file runs that launches MATLAB Compiler (mcc).

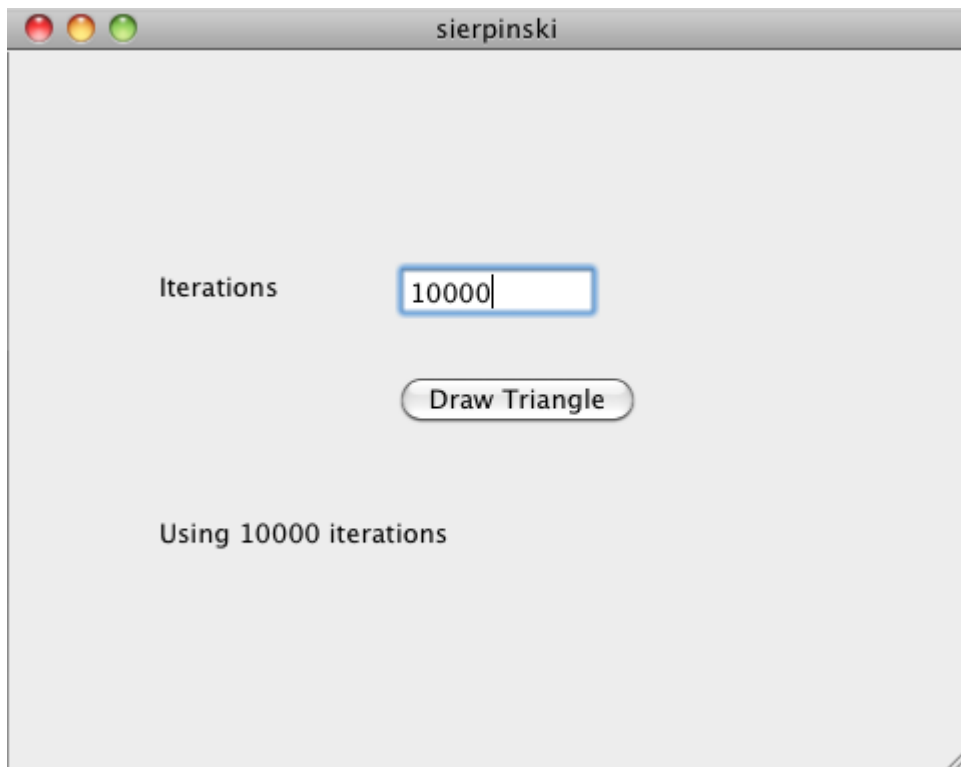
## Running the Sierpinski Application

Run the **Sierpinski** application from the build output directory. The following GUI appears:

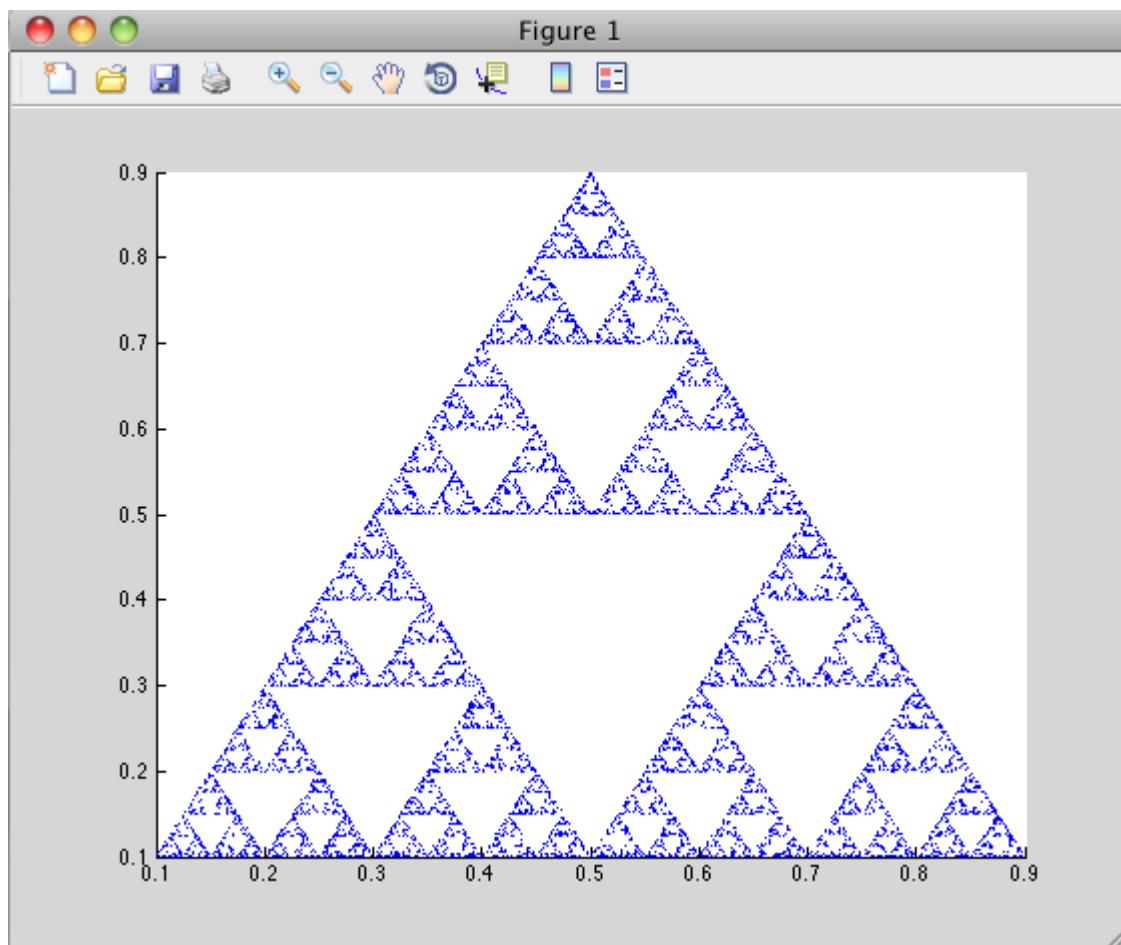


### MATLAB Sierpinski Function Implemented in the Mac Cocoa Environment

- 1 In the **Iterations** field, enter an integer such as 10000:



- 2 Click **Draw Triangle**. The following figure appears:





# Deployment Process

---

This chapter tells you how to deploy compiled MATLAB code to end users.

- “About the MATLAB Runtime” on page 3-2
- “Use Parallel Computing Toolbox in Deployed Applications” on page 3-4
- “Deploy Applications on Network Drives” on page 3-7
- “MATLAB Compiler SDK Deployment Messages” on page 3-8

## About the MATLAB Runtime

<b>In this section...</b>
---------------------------

“How is MATLAB Runtime Different from MATLAB?” on page 3-2
--

“Performance Considerations for MATLAB Runtime” on page 3-2
---

The MATLAB Runtime is a standalone set of shared libraries, MATLAB code, and other files that enables the execution of MATLAB files on computers without an installed version of MATLAB. Applications that use artifacts built with MATLAB Compiler SDK require access to an appropriate version of the MATLAB Runtime to run.

End-users of compiled artifacts without access to MATLAB must install the MATLAB Runtime on their computers or know the location of a network-installed MATLAB Runtime. The installers generated by the compiler apps may include the MATLAB Runtime installer. If you compiled your artifact using `mcc`, you should direct your end-users to download the MATLAB Runtime installer from the website <https://www.mathworks.com/products/compiler/mcr>.

For more information, see “Install and Configure MATLAB Runtime”.

### How is MATLAB Runtime Different from MATLAB?

The MATLAB Runtime differs from MATLAB in several important ways:

- In MATLAB Runtime, MATLAB files are encrypted and immutable.
- MATLAB has a desktop graphical interface. MATLAB Runtime has all the MATLAB functionality without the graphical interface.
- MATLAB Runtime is version-specific. You must run your applications with the version of MATLAB Runtime associated with the version of MATLAB Compiler SDK with which it was created. For example, if you compiled an application using release R2020b of MATLAB, end users must have version 9.9 or later of MATLAB Runtime installed. Use `mcrversion` to return the version number of MATLAB Runtime.
- The MATLAB paths in a MATLAB Runtime instance are fixed and cannot be changed. To change them, you must first customize them within MATLAB.

### Performance Considerations for MATLAB Runtime

MATLAB Compiler SDK was designed to work with a large range of applications that use the MATLAB programming language. Because of this, run-time libraries are large.

Since MATLAB Runtime provides full support for the MATLAB language, including the Java programming language, starting a compiled application takes approximately the same amount of time as starting MATLAB. The amount of resources consumed by MATLAB Runtime is necessary in order to retain the power and functionality of a full version of MATLAB.

Calls into MATLAB Runtime are serialized, so calls into MATLAB Runtime are threadsafe. This can impact performance.

### See Also

`mcrversion` | `compiler.runtime.download`

## **Related Examples**

- “Install and Configure MATLAB Runtime”

## Use Parallel Computing Toolbox in Deployed Applications

An application that uses the Parallel Computing Toolbox can use cluster profiles that are in your MATLAB preferences folder. To find this folder, use `prefdir`.

For instance, when you create a standalone application, all of the profiles available in your **Cluster Profile Manager** will be available in the application.

Your application can also use a cluster profile given in an external file. To enable your application to use this file, you can either:

- 1 Link to the file within your code.
- 2 Pass the location of the file at run time.

### Export Cluster Profile

To export a cluster profile to an external file:

- 1 In the Home tab, in the **Environment** section, select **Parallel > Create and Manage Clusters**.
- 2 In the **Cluster Profile Manager** dialog, select a profile, and in the **Manage** section, click **Export**.

### Link to Parallel Computing Toolbox Profile Within Your Code

To enable your application to use a cluster profile given in an external file, you can link to the file from your code. In this example, you will use absolute paths, relative paths, and the MATLAB search path to link to cluster profiles. Note that since each link is specified before you compile, you must ensure that each link does not change.

To set the cluster profile for your application, you can use the `setmcruserdata` function.

As your MATLAB preferences folder is bundled with your application, any relative links to files within the folder will always work. In your application code, you can use the `myClusterProfile.mlsettings` file found within the MATLAB preferences folder.

```
mpSettingsPath = fullfile(prefdir, 'myClusterProfile.mlsettings');  
setmcruserdata('ParallelProfile', mpSettingsPath);
```

The function `fullfile` gives the absolute path for the external file. The argument given by `mpSettingsPath` must be an absolute path. If the user of your application has a cluster profile located on their file system at an absolute path that will not change, link to it directly:

```
mpSettingsPath = '/path/to/myClusterProfile.mlsettings';  
setmcruserdata('ParallelProfile', mpSettingsPath);
```

This is a good practice if the cluster profile is centrally managed for your application. If the user of your application has a cluster profile that is held locally, you can expand a relative path to it from the current working directory:

```
mpSettingsPath = fullfile(pwd, '../rel/path/to/myClusterProfile.mlsettings');  
setmcruserdata('ParallelProfile', mpSettingsPath);
```

This is a good practice if the user of your standalone application should supply their own cluster profile. Any files that you add to your application at compilation are added to the MATLAB search

path. Therefore, you can also bundle a cluster profile that is held externally with your application. First, use `which` to get the absolute path to the cluster profile. Then, link to it.

```
mpSettingsPath = which('myClusterProfile.mlsettings');
setmcruserdata('ParallelProfile', mpSettingsPath);
```

Finally, compile at the command line and add the cluster profile.

```
mcc -a /path/to/myClusterProfile.mlSettings -m myApp.m;
```

Note that to run your application before you compile, you must manually add `/path/to/` to your MATLAB search path.

## Pass Parallel Computing Toolbox Profile at Run Time

If the user of your application *myApp* has a cluster profile that is selected at run time, you can specify this at the command line.

```
myApp -mcruserdata ParallelProfile:/path/to/myClusterProfile.mlsettings
```

Note that when you use the `setmcruserdata` function in your code, you override the use of the `-mcruserdata` flag.

## Switch Between Cluster Profiles in Deployed Applications

When you use the `setmcruserdata` function, you remove the ability to use any of the profiles available in your Cluster Profile Manager. To re-enable the use of the profiles in **Cluster Profile Manager**, use the `parallel.mlSettings` file.

```
mpSettingsPath = '/path/to/myClusterProfile.mlsettings';
setmcruserdata('ParallelProfile', mpSettingsPath);

% SOME APPLICATION CODE

origSettingsPath = fullfile(prefdir, 'parallel.mlsettings');
setmcruserdata('ParallelProfile', origSettingsPath);

% MORE APPLICATION CODE
```

## Sample C Code to Load Cluster Profile

You can call the `mcruserdata` function natively in C and C++ applications built with MATLAB Compiler SDK.

```
mxArray *key = mxCreateString("ParallelProfile");
mxArray *value = mxCreateString("/path/to/myClusterProfile.mlsettings");
if (!setmcruserdata(key, value))
{
    fprintf(stderr,
            "Could not set MCR user data: \n %s ",
            mclGetLastErrorMessage());
    return -1;
}
```

## **See Also**

setmcruserdata | getmcruserdata

## **Related Examples**

- “Using MATLAB Runtime User Data Interface”
- “Specify Parallel Computing Toolbox Profile in .NET Application”
- “Specify Parallel Computing Toolbox Profile in Java Application”

## Deploy Applications on Network Drives

You can deploy a compiled application to a network drive so that it can be accessed by all network users without having them install MATLAB Runtime on their individual machines.

---

**Note** There is no need to perform these steps on a Linux system.

The component registration is in support of Excel® add-ins and COM components, which both run on Windows only.

Distributing to a Linux network file system is exactly the same as distributing to a local file system. You only need to set up the LD\_LIBRARY\_PATH or use scripts which points to the MATLAB Runtime installation. For more information, see “Set MATLAB Runtime Path for Deployment” on page 7-4.

---

- 1 On any Windows machine, run `mcrinstaller` function to obtain name of the MATLAB Runtime Installer executable.
- 2 Copy the entire MATLAB Runtime installation folder onto a network drive.
- 3 Copy the compiled application into a separate folder in the network drive and add the path `<MATLAB_RUNTIME_INSTALL_DIR>\<ver>\runtime\<arch>` to all client machines. All network users can then execute the application.
- 4 Run `vcredist_x86.exe` on for 32-bit clients; run `vcredist_x64.exe` for 64-bit clients.
- 5 If you are using MATLAB Compiler SDK to create COM objects, register `mwcomutil.dll` on every client machine.

To register the DLLs, at the DOS prompt enter

```
mwregsvr <fully_qualified_pathname\dllname.dll>
```

These DLLs are located in `<MATLAB_RUNTIME_INSTALL_DIR>\<ver>\runtime\<arch>`.

---

**Note** These libraries are automatically registered on the machine on which the installer was run.

---

## **MATLAB Compiler SDK Deployment Messages**

To enable display of MATLAB Compiler SDK deployment messages, see the *MATLAB Desktop Tools and Environment* documentation.



# Distributing Code to an End User

---

## MATLAB Runtime Component Cache and Deployable Archive Embedding

Deployable archive data is automatically embedded directly in shared libraries by default and extracted to a temporary folder.

Automatic embedding enables usage of the MATLAB Runtime component cache features through environment variables.

These variables allow you to specify the following:

- Define the default location where you want the deployable archive to be automatically extracted
- Add diagnostic error printing options that can be used when automatically extracting the deployable archive, for troubleshooting purposes
- Tuning the MATLAB Runtime component cache size for performance reasons.

Use the following environment variables to change these settings.

Environment Variable	Purpose	Notes
MCR_CACHE_ROOT	When set to the location of where you want the deployable archive to be extracted, this variable overrides the default per-user component cache location. This is true for embedded <code>.ctf</code> files only.	On macOS, this variable is ignored in MATLAB R2020a and later. The app bundle contains the files necessary for runtime.
MCR_CACHE_SIZE	When set, this variable overrides the default component cache size.	The initial limit for this variable is 32M (megabytes). This may, however, be changed after you have set the variable the first time. Edit the file <code>.max_size</code> , which resides in the file designated by running the <code>mrcachedir</code> command, with the desired cache size limit.

---

**Note** If you run `mcc` specifying conflicting wrapper and target types, the archive will not be embedded into the generated component. For example, if you run:

```
mcc -W lib:myLib -T link:exe test.m test.c
```

the generated `test.exe` will not have the archive embedded in it, as if you had specified a `-C` option to the command line.

---

**Caution** Do not extract the files within the `.ctf` file and place them individually under version control. Since the `.ctf` file contains interdependent MATLAB functions and data, the files within it must be accessed only by accessing the `.ctf` file. For best results, place the entire `.ctf` file under version control.

---

# Compiler Commands

---

This chapter describes `mcc`, which is the command that invokes the compiler.

## Compiler Tips

<b>In this section...</b>
---------------------------

“Deploying Applications That Call the Java Native Libraries” on page 5-2
--

“Using the VER Function in a Compiled MATLAB Application” on page 5-2
---

### Deploying Applications That Call the Java Native Libraries

If your application interacts with Java, you need to specify the search path for native method libraries by editing `librarypath.txt` and deploying it.

- 1 Copy `librarypath.txt` from `matlabroot/toolbox/local/librarypath.txt`.
- 2 Place `librarypath.txt` in `<mcr_root>/<ver>/toolbox/local`.

`<mcr_root>` refers to the complete path where the MATLAB Runtime library archive files are installed on your machine.

- 3 Edit `librarypath.txt` by adding the folder that contains the native library that your application's Java code needs to load.

### Using the VER Function in a Compiled MATLAB Application

When you use the `VER` function in a compiled MATLAB application, it will perform with the same functionality as if you had called it from MATLAB. However, be aware that when using `VER` in a compiled MATLAB application, only version information for toolboxes which the compiled application uses will be displayed.

# Troubleshooting

---

- “Common Issues” on page 6-2
- “Compilation Failures” on page 6-3
- “Testing Failures” on page 6-5
- “Deployment Failures” on page 6-8
- “Troubleshoot mbuild” on page 6-10
- “Deployed Applications” on page 6-11

## Common Issues

Some of the most common issues encountered when using MATLAB Compiler SDK generated shared libraries are:

- **Compilation fails with an error message.** This can indicate a failure during any one of the internal steps involved in producing the final output.
- **Compilation succeeds but the application does not execute because required DLLs are not found.** All shared libraries required for your standalone executable or shared library are contained in the MATLAB Runtime. Installing the MATLAB Runtime is required for any of the deployment targets.
- **Compilation succeeds, and the resultant file starts to execute but then produces errors and/or generates a crash dump.**
- **The compiled program executes on the machine where it was compiled but not on other machines.**
- **The compiled program executes on some machines and not others.**

## Compilation Failures

You typically compile your MATLAB code on a development machine, test the resulting executable on that machine, and deploy the executable and MATLAB Runtime to a test or customer machine without MATLAB. The compilation process performs dependency analysis on your MATLAB code, creates an encrypted archive of your code and required toolbox code, generates wrapper code, and compiles the wrapper code into an executable. If your application fails to build an executable, the following questions may help you isolate the problem.

### Is your installed compiler supported by MATLAB Compiler SDK?

See the current list of supported compilers at [http://www.mathworks.com/support/compilers/current\\_release/](http://www.mathworks.com/support/compilers/current_release/).

### Are you compiling within or outside of MATLAB?

`mcc` can be invoked from the operating system command line or from the MATLAB prompt. When you run `mcc` inside the MATLAB environment, MATLAB will modify environment variables in its environment as necessary so `mcc` will run. Issues with `PATH`, `LD_LIBRARY_PATH`, or other environment variables seen at the operating system command line are often not seen at the MATLAB prompt. The environment that MATLAB uses for `mcc` can be listed at the MATLAB prompt. For example:

```
>>!set
```

lists the environment on Windows platforms.

```
>>!printenv
```

lists the environment on UNIX platforms. Using this path allows you to use `mcc` from the operating system command line.

### Have you tried to compile any of the C/C++ examples in MATLAB Compiler SDK help?

The source code for all C/C++ examples is provided with MATLAB Compiler SDK and is located in `matlabroot\extern\examples\compilersdk`, where `matlabroot` is the root folder of your MATLAB installation.

### Is your MATLAB object failing to load?

If your MATLAB object fails to load, it is typically a result of the MATLAB Runtime not finding required class definitions.

When working with MATLAB objects that are loaded from a MAT file, remember to include the following statement in your MATLAB function:

```
##function class_constructor
```

Using the `##function` pragma forces dependency analyzer to load needed class definitions, enabling the MATLAB Runtime to successfully load the object.

### If you are compiling a driver application, are you using `mbuild`?

MathWorks recommends and supports using `mbuild` to compile your driver application. `mbuild` is designed and tested to correctly build driver applications. It will ensure that all MATLAB header files

are found by the C/C++ compiler, and that all necessary libraries are specified and found by the linker.

### **Are you trying to compile your driver application using Microsoft Visual Studio or another IDE?**

If you are using an IDE, in addition to linking to the generated export library, you need to include an additional dependency to `mclmcr rt.lib`. This library is provided for all supported Microsoft compilers in `matlabroot\extern\lib\arch\microsoft`.

### **Are you importing the correct versions of import libraries?**

If you have multiple versions of MATLAB installed on your machine, it is possible that an older or incompatible version of the library is referenced. Ensure that the only MATLAB library that you are linking to is `mclmcr rt.lib` and that it is referenced from the appropriate folder.

### **Are you able to compile the matrixdriver example?**

Try following the example “Create a C Shared Library with MATLAB Code”. Typically, if you cannot compile the examples in the documentation, it indicates an issue with the installation of MATLAB or your system compiler.

### **Do you get the MATLAB:I18n:InconsistentLocale Warning?**

The warning message

```
MATLAB:I18n:InconsistentLocale - The system locale setting,  
system_locale_name, is different from the user locale  
setting, user_locale_name
```

indicates a mismatch between locale setting on Microsoft Windows systems. This may affect your ability to display certain characters. For information about changing the locale settings, see your operating system Help.



## Testing Failures

After you have successfully compiled your application, the next step is to test it on a development machine and deploy it on a target machine. The target machine requires MATLAB Runtime to be installed. An installer packaged using `compiler.package.installer` or a `deploytool` app includes all of the files that are required by your application to run, which include the executable, deployable archive, and MATLAB Runtime.

For information on distribution contents for specific application types and platforms, see “Distribute Files to Application Developers”.

Test the application on the development machine by running the application against MATLAB Runtime shipped with MATLAB Compiler SDK. This verifies that library dependencies are correct, that the deployable archive can be extracted, and that all MATLAB code, MEX—files, and support files required by the application have been included in the archive. If you encounter errors testing your application, the following questions may help you isolate the problem.

### **Are you able to execute the application from MATLAB?**

On the development machine, test your application's execution by issuing `!application-name` at the MATLAB prompt. If your application executes within MATLAB but not from outside, this can indicate an issue with the one of the system variables:

- `PATH`
- `LD_LIBRARY_PATH`
- `DYLD_LIBRARY_PATH`

For more information on setting the system library path, see “Set MATLAB Runtime Path for Deployment”.

### **Does the application begin execution and result in MATLAB or other errors?**

Ensure that you included all necessary files when compiling your application (see the `readme.txt` file generated with your compilation for more details).

Functions that are called from your main MATLAB file are automatically included by MATLAB Compiler SDK as are functions included using the `%#function` pragma. However, functions that are not explicitly called, for example through `EVAL`, need to be included at compilation using the `-a` switch of the `mcc` command. Also, any support files like `.mat`, `.txt`, or `.html` files need to be added to the archive with the `-a` switch. For more information, see “Access Files in Packaged Applications”.

There is a limitation on the functionality of MATLAB and associated toolboxes that can be compiled. Check the documentation to see that the functions used in your application's MATLAB files are valid. Check the file `mccExcludedFiles.log` on the development machine. This file lists all functions called from your application that cannot be compiled.

### **Do you have multiple MATLAB versions installed?**

Executables generated using MATLAB Compiler SDK components are designed to run in an environment where multiple versions of MATLAB are installed. Some older versions of MATLAB may not be fully compatible with this architecture.

On Windows, ensure that the *matlabroot*\runtime\win64 of the version of MATLAB in which you are compiling appears ahead of *matlabroot*\runtime\win64 of other versions of MATLAB installed on the PATH environment variable on your machine.

Similarly, on UNIX, ensure that the dynamic library paths (LD\_LIBRARY\_PATH on Linux) match. Do this by comparing the outputs of !printenv at the MATLAB prompt and printenv at the shell prompt. Using this path allows you to use mcc from the operating system command line.

### **If you are testing a shared library and application, did you install MATLAB Runtime?**

All runtime libraries required for any deployment target are contained in MATLAB Runtime. For information on installing MATLAB Runtime, see “Install and Configure MATLAB Runtime”.

### **Do you receive an error message about a missing DLL?**

Error messages indicating missing DLLs such as mclmcrrtX\_XX.dll or mclmcrrtX\_XX.so are generally caused by an incorrect installation of MATLAB Runtime. For information on installing MATLAB Runtime, see “Install and Configure MATLAB Runtime”.

It is also possible that MATLAB Runtime is installed correctly, but the PATH,LD\_LIBRARY\_PATH, or DYLD\_LIBRARY\_PATH variable is set incorrectly. For information on setting environment variables, see “Set MATLAB Runtime Path for Deployment”.

---

**Caution** Do not solve these problems by moving libraries or other files within the MATLAB Runtime folder structure. The MATLAB Runtime system is designed to accommodate different MATLAB Runtime versions operating on the same machine. The folder structure is an important part of this feature.

---

### **Are you receiving errors when trying to run the shared library application?**

Calling MATLAB Compiler SDK generated shared libraries requires correct initialization and termination in addition to library calls themselves. For information on calling shared libraries, see “Call MATLAB Compiler SDK API Functions from C/C++” on page 2-17.

Some key points to consider to avoid errors at run time:

- Ensure that the calls to `mclinitializeApplication` and `libnameInitialize` are successful. The first function enables construction of MATLAB Runtime instances. The second creates the MATLAB Runtime instance required by the library named *libname*. If these calls are not successful, your application will not execute.
- Do not use any `mw-` or `mx-` functions before calling `mclinitializeApplication`. This includes static and global variables that are initialized at program start. Referencing `mw-` or `mx-` functions before initialization results in undefined behavior.
- Do not re-initialize (call `mclinitializeApplication`) after terminating it with `mclTerminateApplication`. The `mclinitializeApplication` and `libnameInitialize` functions should be called only once.
- Ensure that you do not have any library calls after `mclTerminateApplication`.
- Ensure that you are using the correct syntax to call the library and its functions.

### **Does your system’s graphics card support the graphics application?**

In situations where the existing hardware graphics card does not support the graphics application, use software OpenGL<sup>®</sup>. OpenGL libraries are visible for an application by appending

---

*matlabroot/sys/opengl/lib/arch* to the library path. For example, on Linux, enter the following in a Bash shell:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:matlabroot/sys/opengl/lib/glnxa64
```

For more information on setting environment variables, see “Set MATLAB Runtime Path for Deployment”.

### **Is OpenGL properly installed on your system?**

When searching for OpenGL libraries, MATLAB Runtime first looks on the system library path. If OpenGL is not found there, it uses the LD\_LIBRARY\_PATH environment variable to locate the libraries. If you are getting failures due to the OpenGL libraries not being found, you can append the location of the OpenGL libraries to the LD\_LIBRARY\_PATH environment variable. For example, on Linux, enter the following in a Bash shell:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:matlabroot/sys/opengl/lib/glnxa64
```

For more information on setting environment variables, see “Set MATLAB Runtime Path for Deployment”.

## Deployment Failures

After the application is working on the test machine, failures can be isolated in end-user deployment. The end users of your application need to install MATLAB Runtime on their machines. MATLAB Runtime includes a set of shared libraries that provides support for all features of MATLAB. If your application fails during end-user deployment, the following questions may help you isolate the problem.

---

**Note** There are a number of reasons why your application might not deploy to end users, after running successfully in a test environment. For a detailed list of guidelines for writing MATLAB code that can be consumed by end users, see “Write Deployable MATLAB Code”

---

### Is MATLAB Runtime installed?

Installing MATLAB Runtime is required for any of the deployment targets. For details, see “Install and Configure MATLAB Runtime”.

### If running on Linux or macOS, did you update the dynamic library path after installing MATLAB Runtime?

For information on setting the path on a deployment machine after installing MATLAB Runtime, see “Set MATLAB Runtime Path for Deployment”.

### Do you receive an error message about a missing DLL?

Error messages indicating missing DLLs such as `mclmcr rtX_XX.dll` or `mclmcr rtX_XX.so` are generally caused by an incorrect installation of MATLAB Runtime. For information on installing MATLAB Runtime, see “Install and Configure MATLAB Runtime”.

It is also possible that MATLAB Runtime is installed correctly, but the `PATH`, `LD_LIBRARY_PATH`, or `DYLD_LIBRARY_PATH` variable is set incorrectly. For information on setting environment variables, see “Set MATLAB Runtime Path for Deployment”.

---

**Caution** Do not solve these problems by moving libraries or other files within the MATLAB Runtime folder structure. The MATLAB Runtime system is designed to accommodate different MATLAB Runtime versions operating on the same machine. The folder structure is an important part of this feature.

---

### Do you have write access to the necessary folders?

The first operation attempted by an application with compiled MATLAB code is extraction of the deployable archive. If the archive is not extracted, the application cannot access the compiled MATLAB code and the application fails.

There are three possible folders where the deployable archive is extracted:

- If the deployable archive is embedded and you are using the default environment settings, the archive extracts into the current user’s temp folder.
- If the deployable archive is embedded and you set the environment variable `MCR_CACHE_ROOT`, the archive extracts into the folder specified by `MCR_CACHE_ROOT`.

- If the deployable archive is not embedded, the archive extracts into the current folder of the component.

## Troubleshoot mbuild

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

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

**Directory or File Not Writeable.** If a destination folder 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.** 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 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 in the options file. For Microsoft Visual Studio®, for example, make sure to run `vcvars32.bat` (MSVC 6.x and earlier) or `vsvars32.bat` (MSVC 8.x and later).

**mbuild Not a Recognized Command.** If `mbuild` is not recognized, verify that `matlabroot\bin` is in your path. On UNIX, it may be necessary to rehash.

**mbuild Works from Shell But Not from MATLAB (UNIX).** If the command

```
mcc -m hello
```

works from the UNIX command prompt but not from the MATLAB prompt, you may have a problem with your `.bashrc` file. When MATLAB launches a new shell to perform compilations, it executes the `.bashrc` script. If this script causes unexpected changes to the `PATH` environment variable, an error may occur. You can test this before starting MATLAB by performing the following:

```
export SHELL /bin/sh
```

If this works correctly, then you should check your `.bashrc` file for problems setting the `PATH` environment variable.

**Internal Error when Using mbuild -setup (Windows).** Some antivirus software packages may conflict with the `mbuild-setup` process. If you get an error message during `mbuild -setup` of the following form

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

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

**Verification of mbuild Fails.** If none of the previous solutions address your difficulty with `mbuild`, contact MathWorks Technical Support.

## Deployed Applications

**Checking access to X display <IP-address>:0.0 . . . If no response hit ^C and fix host or access control to host. Otherwise, checkout any error messages that follow and fix . . . Successful. . . .** This message can be ignored.

**??? Error: File: /home/username/<MATLAB file\_name>Line: 1651 Column: 8 Arguments to IMPORT must either end with ".\*" or else specify a fully qualified class name: "<class\_name>" fails this test.** The import statement is referencing a Java class (<class\_name>) that MATLAB Compiler SDK (if the error occurs at compile time) or MATLAB Runtime (if the error occurs at run time) cannot find. To work around this, ensure that the JAR file that contains the Java class is stored in a folder that is on the Java class path. (See *matlabroot/toolbox/local/classpath.txt* for the class path.) If the error occurs at run time, the classpath is stored in *matlabroot/toolbox/local/classpath.txt* when running on the development machine. It is stored in <MATLAB\_RUNTIME\_INSTALL\_DIR>/toolbox/local/classpath.txt when running on a target machine.

**Undefined function or variable 'matlabrc'.** When MATLAB or the MATLAB Runtime starts, they attempt to execute the MATLAB file *matlabrc.m*. This message means that this file cannot be found. To work around this, try each of these suggestions in this order:

- Ensure that your application runs in MATLAB (uncompiled) without this error.
- Ensure that MATLAB starts up without this error.
- Verify that the generated deployable archive contains a file called *matlabrc.m*.
- Verify that the generated code (in the *\*\_mcc\_component\_data.c\** file) adds the deployable archive folder containing *matlabrc.m* to the MATLAB Runtime path.
- Delete the *\*\_mcr* folder and rerun the application.
- Recompile the application.

**Error: library mclmcrX\_XX.dll not found.** This error can occur for the following reasons:

- The machine on which you are trying to run the application uses a different, incompatible version of MATLAB Runtime than the one the application was originally built with.
- You are not running a version of MATLAB Compiler SDK compatible with the MATLAB Runtime version the application was built with.

To solve this problem, on the deployment machine, install the same version of MATLAB or MATLAB Runtime you used to build the application.

**Invalid .NET Framework.\n Either the specified framework was not found or is not currently supported.** This error occurs when the .NET Framework version your application is specifying (represented by *n*) is not supported by the current version of MATLAB Compiler SDK.

**System.AccessViolationException: Attempted to read or write protected memory.** The message:

```
System.ArgumentException: Generate Queries
    threw General Exception:
System.AccessViolationException: Attempted to
    read or write protected memory.
This is often an indication that other memory is corrupt.
```

indicates a library initialization error caused by a Microsoft Visual Studio project linked against a `MCLMCRRTX_XX.DLL` placed outside *matlabroot*.



# Reference Information

---

- “MATLAB Runtime Path Settings for Development and Testing” on page 7-2
- “Set MATLAB Runtime Path for Deployment” on page 7-4
- “MATLAB Compiler SDK Licensing” on page 7-8
- “Deployment Product Terms” on page 7-9

## MATLAB Runtime Path Settings for Development and Testing

### In this section...

“Path for Java Development on All Platforms” on page 7-2

“Path Modifications Required for Accessibility” on page 7-2

“Windows Settings for Development and Testing” on page 7-2

“Linux Settings for Development and Testing” on page 7-2

“OS X Settings for Development and Testing” on page 7-2

### Path for Java Development on All Platforms

There are additional requirements when programming in the Java programming language. For more information see “Configure Your Environment for Generating Java Packages”.

### Path Modifications Required for Accessibility

In order to use some screen-readers or assistive technologies, such as JAWS®, you must add the following DLLs to your Windows path:

```
matlabroot\sys\java\jre\arch\jre\bin\JavaAccessBridge.dll  
matlabroot\sys\java\jre\arch\jre\bin\WindowsAccessBridge.dll
```

You may not be able to use such technologies without doing so.

### Windows Settings for Development and Testing

When programming with compiled MATLAB code, add the following folder to your system PATH environment variable:

```
matlabroot\runtime\win32|win64
```

### Linux Settings for Development and Testing

Add the following platform-specific folders to your dynamic library path.

---

**Note** For readability, the following commands appear on separate lines, but you must enter each `setenv` command on one line.

---

```
setenv LD_LIBRARY_PATH  
matlabroot/runtime/glnxa64:  
matlabroot/bin/glnxa64:  
matlabroot/sys/os/glnxa64:  
matlabroot/sys/opengl/lib/glnxa64
```

### OS X Settings for Development and Testing

Add the following platform-specific folders to your dynamic library path.

**Note** For readability, the following commands appear on separate lines, but you must enter each `setenv` command on one line.

---

```
setenv DYLD_LIBRARY_PATH  
  matlabroot/runtime/maci64:  
  matlabroot/bin/maci64:  
  matlabroot/sys/os/maci64:
```

## Set MATLAB Runtime Path for Deployment

### In this section...

“Library Path Environment Variables and MATLAB Runtime Folders” on page 7-4

“Windows” on page 7-5

“Linux” on page 7-5

“macOS” on page 7-6

“Set Path Permanently on UNIX” on page 7-6

Applications generated with MATLAB Compiler or MATLAB Compiler SDK use the system library path to locate the MATLAB Runtime libraries. The MATLAB Runtime installer for Windows automatically sets the library path during installation, but on Linux or macOS you must add the libraries manually. After you install MATLAB Runtime, add the run-time folders to the system library path according to the instructions for your operating system and shell environment.

Alternatively, you can pass the location of MATLAB Runtime as an input to the associated shell script (`run_application.sh`) on Linux or macOS to launch an application.

### Note

- Your library path may contain multiple versions of MATLAB Runtime. Applications launched without using the shell script use the first version listed in the path.
- Save the value of your current library path as a backup before modifying it.
- If you are using a network install of MATLAB Runtime, see “Run Applications Using a Network Installation of MATLAB Runtime”.

## Library Path Environment Variables and MATLAB Runtime Folders

Operating System	Environment Variable	Directories
Windows	PATH	<MATLAB_RUNTIME_INSTALL_DIR>\runtime\<arch>
Linux	LD_LIBRARY_PATH	<MATLAB_RUNTIME_INSTALL_DIR>/runtime/glnxa64 <MATLAB_RUNTIME_INSTALL_DIR>/bin/glnxa64 <MATLAB_RUNTIME_INSTALL_DIR>/sys/os/glnxa64 <MATLAB_RUNTIME_INSTALL_DIR>/extern/bin/glnxa64
macOS	DYLD_LIBRARY_PATH	<MATLAB_RUNTIME_INSTALL_DIR>/runtime/maci64 <MATLAB_RUNTIME_INSTALL_DIR>/bin/maci64 <MATLAB_RUNTIME_INSTALL_DIR>/sys/os/maci64 <MATLAB_RUNTIME_INSTALL_DIR>/extern/bin/maci64

## Windows

The MATLAB Runtime installer for Windows automatically sets the library path during installation. If you do not use the installer, complete the following steps to set the PATH environment variable permanently.

- 1 Run `C:\Windows\System32\SystemPropertiesAdvanced.exe` and click the **Environment Variables...** button.
- 2 Select the system variable Path and click **Edit...**

---

**Note** If you do not have administrator rights on the machine, select the user variable Path instead of the system variable.

---

- 3 Click **New** and add the folder `<MATLAB_RUNTIME_INSTALL_DIR>\runtime\<arch>`.

For example, if you are using MATLAB Runtime R2022b located in the default installation folder on 64-bit Windows, add `C:\Program Files\MATLAB\MATLAB Runtime\R2022b\runtime\win64`.

- 4 Click **OK** to apply the change.

---

**Note** If the path contains multiple versions of MATLAB Runtime, applications use the first version listed in the path.

---

## Linux

For information on setting environment variables in shells other than Bash, see your shell documentation.

### Bash Shell

- 1 Display the current value of LD\_LIBRARY\_PATH in the terminal.  
`echo $LD_LIBRARY_PATH`
- 2 Append the MATLAB Runtime folders to the LD\_LIBRARY\_PATH variable for the current session.

```
export LD_LIBRARY_PATH="${LD_LIBRARY_PATH:+${LD_LIBRARY_PATH}:}\
<MATLAB_RUNTIME_INSTALL_DIR>/runtime/glnxa64:\
<MATLAB_RUNTIME_INSTALL_DIR>/bin/glnxa64:\
<MATLAB_RUNTIME_INSTALL_DIR>/sys/os/glnxa64:\
<MATLAB_RUNTIME_INSTALL_DIR>/extern/bin/glnxa64"
```

---

**Note** If you require Mesa Software OpenGL rendering to resolve low level graphics issues, add the folder `<MATLAB_RUNTIME_INSTALL_DIR>/sys/opengl/lib/glnxa64` to the path. For details, see “Resolving Low-Level Graphics Issues”.

---

For example, if you are using MATLAB Runtime R2022b located in the default installation folder, use the following command:

```
export LD_LIBRARY_PATH="${LD_LIBRARY_PATH:+${LD_LIBRARY_PATH}:}\
/usr/local/MATLAB/MATLAB_Runtime/R2022b/runtime/glnxa64:\
```

```
/usr/local/MATLAB/MATLAB_Runtime/R2022b/bin/glnxa64:\
/usr/local/MATLAB/MATLAB_Runtime/R2022b/sys/os/glnxa64:\
/usr/local/MATLAB/MATLAB_Runtime/R2022b/extern/bin/glnxa64"
```

- 3 Display the new value of LD\_LIBRARY\_PATH to ensure the path is correct.

```
echo $LD_LIBRARY_PATH
```

- 4 Type `ldd --version` to check your version of GNU® C library (glibc). If the version displayed is 2.17 or lower, add `<MATLAB_RUNTIME_INSTALL_DIR>/bin/glnxa64/glibc-2.17_shim.so` to the LD\_PRELOAD environment variable using the following command:

```
export LD_PRELOAD="${LD_PRELOAD:+${LD_PRELOAD}:}\
<MATLAB_RUNTIME_INSTALL_DIR>/bin/glnxa64/glibc-2.17_shim.so"
```

- 5 To make these changes permanent, see “Set Path Permanently on UNIX” on page 7-6.

## macOS

- 1 Display the current value of DYLD\_LIBRARY\_PATH in the terminal.

```
echo $DYLD_LIBRARY_PATH
```

- 2 Append the MATLAB Runtime folders to the DYLD\_LIBRARY\_PATH variable for the current session.

```
export DYLD_LIBRARY_PATH="${DYLD_LIBRARY_PATH:+${DYLD_LIBRARY_PATH}:}\
<MATLAB_RUNTIME_INSTALL_DIR>/runtime/maci64:\
<MATLAB_RUNTIME_INSTALL_DIR>/bin/maci64:\
<MATLAB_RUNTIME_INSTALL_DIR>/sys/os/maci64:\
<MATLAB_RUNTIME_INSTALL_DIR>/extern/bin/maci64"
```

For example, if you are using MATLAB Runtime R2022b located in the default installation folder, use the following command:

```
export DYLD_LIBRARY_PATH="${DYLD_LIBRARY_PATH:+${DYLD_LIBRARY_PATH}:}\
/Applications/MATLAB/MATLAB_Runtime/R2022b/runtime/maci64:\
/Applications/MATLAB/MATLAB_Runtime/R2022b/bin/maci64:\
/Applications/MATLAB/MATLAB_Runtime/R2022b/sys/os/maci64:\
/Applications/MATLAB/MATLAB_Runtime/R2022b/extern/bin/maci64"
```

- 3 Display the value of DYLD\_LIBRARY\_PATH to ensure the path is correct.

```
echo $DYLD_LIBRARY_PATH
```

- 4 To make these changes permanent, see “Set Path Permanently on UNIX” on page 7-6.

## Set Path Permanently on UNIX

---

**Caution** The MATLAB Runtime libraries may conflict with other applications that use the library path. In this case, set the path only for the current session, or run MATLAB Compiler SDK applications using the generated shell script.

---

To set an environment variable at login on Linux or macOS, append the `export` command to the shell configuration file `~/.bash_profile` in a Bash shell or `~/.zprofile` in a Zsh shell.

To determine your current shell environment, type `echo $SHELL`.

## See Also

### More About

- [“Install and Configure MATLAB Runtime”](#)
- [“Run Applications Using a Network Installation of MATLAB Runtime”](#)
- [“Change Environment Variable for Shell Command”](#)

## MATLAB Compiler SDK Licensing

### Use MATLAB Compiler SDK Licenses for Development

You can run the MATLAB Compiler SDK compiler from the MATLAB command prompt or the system prompt.

MATLAB Compiler SDK uses a lingering license. This means that when the MATLAB Compiler SDK license is checked out, a timer is started. When that timer reaches 30 minutes, the license key is returned to the license pool. The license key will not be returned until that 30 minutes is up, regardless of whether `mcc` has exited or not.

Each time a compiler command is issued, the timer is reset.

### Running MATLAB Compiler SDK in MATLAB Mode

When you run MATLAB Compiler SDK from “inside” of the MATLAB environment, that is, you run `mcc` from the MATLAB command prompt, you hold the MATLAB Compiler SDK license as long as MATLAB remains open. To give up the MATLAB Compiler SDK license, exit MATLAB.

### Running MATLAB Compiler SDK in Standalone Mode

If you run MATLAB Compiler SDK from a DOS or UNIX prompt, you are running from “outside” of MATLAB. In this case, MATLAB Compiler SDK

- Does not require MATLAB to be running on the system where MATLAB Compiler SDK is running
- Gives the user a dedicated 30-minute time allotment during which the user has complete ownership over a license to MATLAB Compiler SDK

Each time a user requests MATLAB Compiler SDK, the user begins a 30-minute time period as the sole owner of the MATLAB Compiler SDK license. Anytime during the 30-minute segment, if the same user requests MATLAB Compiler SDK, the user gets a new 30-minute allotment. When the 30-minute interval has elapsed, if a different user requests MATLAB Compiler SDK, the new user gets the next 30-minute interval.

When a user requests MATLAB Compiler SDK and a license is not available, the user receives the message

```
Error: Could not check out a Compiler License.
```

This message is given when no licenses are available. As long as licenses are available, the user gets the license and no message is displayed. The best way to guarantee that all MATLAB Compiler SDK users have constant access to MATLAB Compiler SDK is to have an adequate supply of licenses for your users.



## Deployment Product Terms

### A

*Add-in* — A Microsoft Excel add-in is an executable piece of code that can be actively integrated into a Microsoft Excel application. Add-ins are front-ends for COM components, usually written in some form of Microsoft Visual Basic®.

*Application program interface (API)* — A set of classes, methods, and interfaces that is used to develop software applications. Typically an API is used to provide access to specific functionality. See *MWArray*.

*Application* — An end user-system into which a deployed functions or solution is ultimately integrated. Typically, the end goal for the deployment customer is integration of a deployed MATLAB function into a larger enterprise environment application. The deployment products prepare the MATLAB function for integration by wrapping MATLAB code with enterprise-compatible source code, such as C, C++, C# (.NET), F#, and Java code.

*Assembly* — An executable bundle of code, especially in .NET.

### B

*Binary* — See *Executable*.

*Boxed Types* — Data types used to wrap opaque C structures.

*Build* — See *Compile*.

### C

*Class* — A user-defined type used in C++, C#, and Java, among other object-oriented languages, that is a prototype for an object in an object-oriented language. It is analogous to a derived type in a procedural language. A class is a set of objects which share a common structure and behavior. Classes relate in a class hierarchy. One class is a specialization (a subclass) of another (one of its *superclasses*) or comprises other classes. Some classes use other classes in a client-server relationship. Abstract classes have no members, and concrete classes have one or more members. Differs from a MATLAB class

*Compile* — In MATLAB Compiler and MATLAB Compiler SDK, to compile MATLAB code involves generating a binary that wraps around MATLAB code, enabling it to execute in various computing environments. For example, when MATLAB code is compiled into a Java package, a Java wrapper provides Java code that enables the MATLAB code to execute in a Java environment.

*COM component* — In MATLAB Compiler, the executable back-end code behind a Microsoft Excel add-in. In MATLAB Compiler SDK, an executable component, to be integrated with Microsoft COM applications.

*Console application* — Any application that is executed from a system command prompt window.

### D

*Data Marshaling* — Data conversion, usually from one type to another. Unless a MATLAB deployment customer is using type-safe interfaces, data marshaling—as from mathematical data types to MathWorks data types such as represented by the *MWArray* API—must be performed manually, often at great cost.

*Deploy* — The act of integrating MATLAB code into a larger-scale computing environment, usually to an enterprise application, and often to end users.

*Deployable archive* — The deployable archive is embedded by default in each binary generated by MATLAB Compiler or MATLAB Compiler SDK. It houses the deployable package. All MATLAB-based content in the deployable archive uses the Advanced Encryption Standard (AES) cryptosystem. See “Additional Details”.

*DLL* — Dynamic link library. Microsoft's implementation of the shared library concept for Windows. Using DLLs is much preferred over the previous technology of static (or non-dynamic) libraries, which had to be manually linked and updated.

## **E**

*Empties* — Arrays of zero (0) dimensions.

*Executable* — An executable bundle of code, made up of binary bits (zeros and ones) and sometimes called a *binary*.

## **F**

*Fields* — For this definition in the context of MATLAB Data Structures, see *Structs*.

*Fields and Properties* — In the context of .NET, *Fields* are specialized classes used to hold data. *Properties* allow users to access class variables as if they were accessing member fields directly, while actually implementing that access through a class method.

## **I**

*Integration* — Combining deployed MATLAB code's functionality with functionality that currently exists in an enterprise application. For example, a customer creates a mathematical model to forecast trends in certain commodities markets. In order to use this model in a larger-scale financial application (one written with the Microsoft .NET Framework, for instance) the deployed financial model must be integrated with existing C# applications, run in the .NET enterprise environment.

*Instance* — For the definition of this term in context of MATLAB Production Server™ software, see *MATLAB Production Server Server Instance*.

## **J**

*JAR* — Java archive. In computing software, a JAR file (or Java Archive) aggregates many files into one. Software developers use JARs to distribute Java applications or libraries, in the form of classes and associated metadata and resources (text, images, etc.). Computer users can create or extract JAR files using the `jar` command that comes with a Java Development Kit (JDK).

*Java-MATLAB Interface* — Known as the *JMI Interface*, this is the Java interface built into MATLAB software.

*JDK* — The Java Development Kit is a product which provides the environment required for programming in Java.

*JMI Interface* — see *Java-MATLAB Interface*.

*JRE* — Java Run-Time Environment is the part of the Java Development Kit (JDK) required to run Java programs. It comprises the Java Virtual Machine, the Java platform core classes, and supporting files.

It does not include the compiler, debugger, or other tools present in the JDK™. The JRE™ is the smallest set of executables and files that constitute the standard Java platform.

## M

*Magic Square* — A square array of integers arranged so that their sum is the same when added vertically, horizontally, or diagonally.

*MATLAB Runtime* — An execution engine made up of the same shared libraries. MATLAB uses these libraries to enable the execution of MATLAB files on systems without an installed version of MATLAB.

*MATLAB Runtime singleton* — See *Shared MATLAB Runtime instance*.

*MATLAB Runtime workers* — A MATLAB Runtime session. Using MATLAB Production Server software, you have the option of specifying more than one MATLAB Runtime session, using the `--num-workers` options in the server configurations file.

*MATLAB Production Server Client* — In the MATLAB Production Server software, clients are applications written in a language supported by MATLAB Production Server that call deployed functions hosted on a server.

*MATLAB Production Server Configuration* — An instance of the MATLAB Production Server containing at least one server and one client. Each configuration of the software usually contains a unique set of values in the server configuration file, `main_config` (MATLAB Production Server).

*MATLAB Production Server Server Instance* — A logical server configuration created using the `mps-new` command in MATLAB Production Server software.

*MATLAB Production Server Software* — Product for server/client deployment of MATLAB programs within your production systems, enabling you to incorporate numerical analytics in enterprise applications. When you use this software, web, database, and enterprise applications connect to MATLAB programs running on MATLAB Production Server via a lightweight client library, isolating the MATLAB programs from your production system. MATLAB Production Server software consists of one or more servers and clients.

*Marshaling* — See *Data Marshaling*.

*mbuild* — MATLAB Compiler SDK command that compiles and links C and C++ source files into standalone applications or shared libraries. For more information, see the `mbuild` function reference page.

*mcc* — The MATLAB command that invokes the compiler. It is the command-line equivalent of using the compiler apps.

*Method Attribute* — In the context of .NET, a mechanism used to specify declarative information to a .NET class. For example, in the context of client programming with MATLAB Production Server software, you specify method attributes to define MATLAB structures for input and output processing.

*mxArray interface* — The MATLAB data type containing all MATLAB representations of standard mathematical data types.

*MWArray interface* — A proxy to `mxArray`. An application program interface (API) for exchanging data between your application and MATLAB. Using `MWArray`, you marshal data from traditional mathematical types to a form that can be processed and understood by MATLAB data type `mxArray`.

There are different implementations of the `MWArray` proxy for each application programming language.

## **P**

*Package* — The act of bundling the deployed MATLAB code, along with the MATLAB Runtime and other files, into an installer that can be distributed to others. The compiler apps place the installer in the `for_redistribution` subfolder. In addition to the installer, the compiler apps generate a number of loose artifacts that can be used for testing or building a custom installer.

*PID File* — See *Process Identification File (PID File)*.

*Pool* — A pool of threads, in the context of server management using MATLAB Production Server software. Servers created with the software do not allocate a unique thread to each client connection. Rather, when data is available on a connection, the required processing is scheduled on a pool, or group, of available threads. The server configuration file option `--num-threads` sets the size of that pool (the number of available request-processing threads) in the master server process.

*Process Identification File (PID File)* — A file that documents informational and error messages relating to a running server instance of MATLAB Production Server software.

*Program* — A bundle of code that is executed to achieve a purpose. Programs usually are written to automate repetitive operations through computer processing. Enterprise system applications usually consist of hundreds or even thousands of smaller programs.

*Properties* — For this definition in the context of .NET, see *Fields and Properties*.

*Proxy* — A software design pattern typically using a class, which functions as an interface to something else. For example, `MWArray` is a proxy for programmers who need to access the underlying type `mxArray`.

## **S**

*Server Instance* — See MATLAB Production Server Server Instance.

*Shared Library* — Groups of files that reside in one space on disk or memory for fast loading into Windows applications. Dynamic-link libraries (DLLs) are Microsoft's implementation of the shared library concept for Microsoft Windows.

*Shared MATLAB Runtime instance* — When using MATLAB Compiler SDK, you can create a shared MATLAB Runtime instance, also known as a singleton. When you invoke MATLAB Compiler with the `-S` option through the compiler (using either `mcc` or a compiler app), a single MATLAB Runtime instance is created for each COM component or Java package in an application. You reuse this instance by sharing it among all subsequent class instances. Such sharing results in more efficient memory usage and eliminates the MATLAB Runtime startup cost in each subsequent class instantiation. All class instances share a single MATLAB workspace and share global variables in the deployed MATLAB files. MATLAB Compiler SDK creates singletons by default for .NET assemblies. MATLAB Compiler creates singletons by default for the COM components used by the Excel add-ins.

*State* — The present condition of MATLAB, or the MATLAB Runtime. MATLAB functions often carry state in the form of variable values. The MATLAB workspace itself also maintains information about global variables and path settings. When deploying functions that carry state, you must often take additional steps to ensure state retention when deploying applications that use such functions.

*Structs* — MATLAB Structures. Structs are MATLAB arrays with elements that you access using textual field designators. Fields are data containers that store data of a specific MATLAB type.

*System Compiler* — A key part of Interactive Development Environments (IDEs) such as Microsoft Visual Studio.

## T

*Thread* — A portion of a program that can run independently of and concurrently with other portions of the program. See *pool* for additional information on managing the number of processing threads available to a server instance.

*Type-safe interface* — An API that minimizes explicit type conversions by hiding the `MWArray` type from the calling application.

## W

*Web Application Archive (WAR)* — In computing, a Web Application Archive is a JAR file used to distribute a collection of JavaServer pages, servlets, Java classes, XML files, tag libraries, and static web pages that together constitute a web application.

*Webfigure* — A MathWorks representation of a MATLAB figure, rendered on the web. Using the WebFigures feature, you display MATLAB figures on a website for graphical manipulation by end users. This enables them to use their graphical applications from anywhere on the web, without the need to download MATLAB or other tools that can consume costly resources.

*Windows Communication Foundation (WCF)* — The Windows Communication Foundation™ is an application programming interface in the .NET Framework for building connected, service-oriented, web-centric applications. WCF is designed in accordance with service oriented architecture principles to support distributed computing where services are consumed by client applications.



# Functions

---

## compiler.build.cppSharedLibrary

Create C++ shared library

### Syntax

```
compiler.build.cppSharedLibrary(FunctionFiles)
compiler.build.cppSharedLibrary(FunctionFiles,Name,Value)
compiler.build.cppSharedLibrary(opts)
results = compiler.build.cppSharedLibrary( ___ )
```

### Description

`compiler.build.cppSharedLibrary(FunctionFiles)` creates a C++ shared library using the MATLAB files specified by `FunctionFiles`. Install a supported C++ compiler before using this function.

`compiler.build.cppSharedLibrary(FunctionFiles,Name,Value)` creates a C++ shared library with options specified using one or more name-value arguments. Options include the interface API, library name, and output directory.

`compiler.build.cppSharedLibrary(opts)` creates a C++ shared library with options specified using a `compiler.build.CppSharedLibraryOptions` object `opts`. You cannot specify any other options using name-value arguments.

`results = compiler.build.cppSharedLibrary( ___ )` returns build information as a `compiler.build.Results` object using any of the input argument combinations in previous syntaxes. The build information consists of the build type, paths to the compiled files, and build options.

### Examples

#### Create C++ Library

Create a C++ shared library using a function file that adds two matrices.

In MATLAB, locate the MATLAB function that you want to deploy as a C++ library. For this example, use the file `addmatrix.m` located in `matlabroot\extern\examples\compilersdk\c_cpp\matrix`.

```
appFile = fullfile(matlabroot,'extern','examples','compilersdk','c_cpp','matrix','addmatrix.m');
```

Build a C++ library using the `compiler.build.cppSharedLibrary` command.

```
compiler.build.cppSharedLibrary(appFile);
```

The build function generates the following files within a folder named `addmatrixcppSharedLibrary` in your current working directory:

- `GettingStarted.html` — HTML file that contains information on integrating your shared library.



- `includedSupportPackages.txt` — Text file that lists all support files included in the library.
- `mccExcludedFiles.log` — Log file that contains a list of any toolbox functions that were not included in the application. For information on non-supported functions, see MATLAB Compiler Limitations.
- `readme.txt` — Readme file that contains information on deployment prerequisites and the list of files to package for deployment.
- `requiredMCRProducts.txt` — Text file that contains product IDs of products required by MATLAB Runtime to run the application.
- `unresolvedSymbols.txt` — Text file that contains information on unresolved symbols.
- `v2\generic_interface\readme.txt` — Readme file that contains information on deployment prerequisites and the list of files to package for deployment.
- `v2\generic_interface\magicsquare.ctf` — Component technology file that contains the deployable archive.

To implement your shared library, see “Implement C++ MATLAB Data API Shared Library with Sample Application”.

## Customize C++ Library

Create a C++ library and customize it using name-value arguments.

For this example, use the file `addmatrix.m` located in `matlabroot\extern\examples\compilersdk\c_cpp\matrix`.

```
appFile = fullfile(matlabroot,'extern','examples','compilersdk','c_cpp','matrix','addmatrix.m');
```

Save the following code in a sample file named `addmatrixSample1.m`:

```
a1 = [1 4 7; 2 5 8; 3 6 9];
a2 = a1;
a = addmatrix(a1, a2);
```

Build a C++ library using the `compiler.build.cppSharedLibrary` command. Use name-value arguments to specify the library name, add a sample file, and use the `mwArray` interface.

```
compiler.build.cppSharedLibrary(appFile,'LibraryName','mwa_addmatrix',...
    'SampleGenerationFiles','addmatrixSample1.m',...
    'Interface','mvarray');
```

The build function creates the following files within a folder named `mwa_addmatrixcppSharedLibrary` in your current working directory:

- `samples\addmatrixSample1_mvarray.cpp` — C++ sample driver file.
- `GettingStarted.html` — File that contains information on integrating your shared library.
- `includedSupportPackages.txt` — Text file that lists all support files included in the library.
- `mwa_addmatrix.cpp` — C++ source code file.
- `mwa_addmatrix.def` — Module-definition file that provides the linker with module information.
- `mwa_addmatrix.dll` — Dynamic-link library file.
- `mwa_addmatrix.exports` — Exports file that contains all nonstatic function names.
- `mwa_addmatrix.h` — C++ header file.

- `mwa_addmatrix.lib` — Import library file.
- `mccExcludedFiles.log` — Log file that contains a list of any toolbox functions that were not included in the application. For information on non-supported functions, see MATLAB Compiler Limitations.
- `readme.txt` — Readme file that contains information on deployment prerequisites and the list of files to package for deployment.
- `requiredMCRProducts.txt` — Text file that contains product IDs of products required by MATLAB Runtime to run the application.
- `unresolvedSymbols.txt` — Text file that contains information on unresolved symbols.

To implement your shared library, see “Implement C++ mxArray API Shared Library with C++ Sample Application”.

### Create Multiple Libraries Using Options Object

Create multiple C++ libraries using a `compiler.build.CppSharedLibraryOptions` object.

For this example, use the file `addmatrix.m` located in `matlabroot\extern\examples\compilersdk\c_cpp\matrix`.

```
appFile = fullfile(matlabroot,'extern','examples','compilersdk','c_cpp','matrix','addmatrix.m');
```

Create a `CppSharedLibraryOptions` object using `appFile`. Use name-value arguments to specify a common output directory, enable debug symbols, and enable verbose output.

```
opts = compiler.build.CppSharedLibraryOptions(appFile,...
    'OutputDir','D:\Documents\MATLAB\work\CppLibraryBatch',...
    'DebugBuild','on',...
    'Verbose','on')
```

```
opts =
```

```
    CppSharedLibraryOptions with properties:
```

```
        Interface: 'matlab-data'
        LibraryVersion: '1.0.0.0'
        SampleGenerationFiles: {}
        FunctionFiles: {'C:\Program Files\MATLAB\R2022b\extern\examples\compilersdk\c_cpp\ad
        DebugBuild: on
        LibraryName: 'addmatrix'
        AdditionalFiles: {}
        AutoDetectDataFiles: on
        SupportPackages: {'autodetect'}
        Verbose: on
        OutputDir: 'D:\Documents\MATLAB\work\CppLibraryBatch'
```

Build the C++ shared library using the `CppSharedLibraryOptions` object.

```
compiler.build.cppSharedLibrary(opts);
```

To create a new library using the function file `subtractmatrix.m` with the same options, use dot notation to modify the `FunctionFiles` argument of the existing `CppSharedLibrary` object before running the build function again.

```
opts.FunctionFiles = fullfile(matlabroot,'extern','examples','compilersdk','c_cpp','matrix','subtractmatrix.m');
compiler.build.cppSharedLibrary(opts);
```

By modifying the `FunctionFiles` argument and recompiling, you can compile multiple libraries using the same options object.

### Get Build Information from C++ Library

Create a C++ library and save information about the build type, compiled files, support packages, and build options to a `compiler.build.Results` object.

Compile using the file `magicsquare.m` located in `matlabroot\extern\examples\compiler`.

```
results = compiler.build.cppSharedLibrary('magicsquare.m')
```

```
results =
```

```
    Results with properties:
```

```
        BuildType: 'cppSharedLibrary'
           Files: {2×1 cell}
IncludedSupportPackages: {}
           Options: [1×1 compiler.build.CppSharedLibraryOptions]
```

The `Files` property contains the paths to the `v2` folder and `GettingStarted.html`.

## Input Arguments

### FunctionFiles — Files implementing MATLAB functions

character vector | string scalar | cell array of character vectors | string array

Files implementing MATLAB functions, specified as a character vector, a string scalar, a string array, or a cell array of character vectors. File paths can be relative to the current working directory or absolute. Files must have a `.m` extension.

Example: `["myfunc1.m", "myfunc2.m"]`

Data Types: `char` | `string` | `cell`

### opts — C++ library build options

`compiler.build.CppSharedLibraryOptions` object

C++ library build options, specified as a `compiler.build.CppSharedLibraryOptions` object.

### Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose `Name` in quotes.*

Example: `'Verbose', 'on'`

### AdditionalFiles — Additional files

character vector | string scalar | cell array of character vectors | string array

Additional files and folders to include in the C++ shared library, specified as a character vector, a string scalar, a string array, or a cell array of character vectors. Paths can be relative to the current working directory or absolute.

Example: `'AdditionalFiles', ["myvars.mat", "data.txt"]`

Data Types: `char` | `string` | `cell`

### **AutoDetectDataFiles — Flag to automatically include data files**

`'on'` (default) | on/off logical value

Flag to automatically include data files, specified as `'on'` or `'off'`, or as numeric or logical 1 (`true`) or 0 (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- If you set this property to `'on'`, then data files that you provide as inputs to certain functions (such as `load` and `fopen`) are automatically included in the shared library.
- If you set this property to `'off'`, then you must add data files to the shared library using the `AdditionalFiles` property.

Example: `'AutoDetectDataFiles', 'off'`

Data Types: `logical`

### **DebugBuild — Flag to enable debug symbols**

`'off'` (default) | on/off logical value

Flag to enable debug symbols, specified as `'on'` or `'off'`, or as numeric or logical 1 (`true`) or 0 (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- If you set this property to `'on'`, then the compiled library contains debug symbols.
- If you set this property to `'off'`, then the compiled library does not contain debug symbols.

Example: `'DebugBuild', 'on'`

Data Types: `logical`

### **Interface — Interface API**

`'matlab-data'` (default) | `'mwstring'`

Interface API, specified as one of the following options:

- `'matlab-data'` — Generate shared libraries using the MATLAB Data API.
- `'mwstring'` — Generate shared libraries using the `mwstring` API.

For more information, see “API Selection for C++ Shared Library”.

Example: `'Interface', 'mwstring'`

### **LibraryName — Name of C++ shared library**

character vector | string scalar

Name of the C++ shared library, specified as a character vector or a string scalar. The default name of the generated library is the first entry of the `FunctionFiles` argument.

Example: 'LibraryName', 'mymagic'

Data Types: char | string

### **OutputDir — Path to output directory**

character vector | string scalar

Path to the output directory where the build files are saved, specified as a character vector or a string scalar. The path can be relative to the current working directory or absolute.

The default name of the build folder is the library name appended with `cppSharedLibrary`.

Example: 'OutputDir', 'D:\Documents\MATLAB\work\mymagiccppSharedLibrary'

Data Types: char | string

### **SampleGenerationFiles — MATLAB sample files**

character vector | string scalar | cell array of character vectors | string array

MATLAB sample files used to generate sample C++ library files for functions included within the library, specified as a character vector, a string scalar, a string array, or a cell array of character vectors. Paths can be relative to the current working directory or absolute. Files must have a `.m` extension. For more information and limitations, see “Sample Driver File Creation”.

Example: 'SampleGenerationFiles', ["sample1.m", "sample2.m"]

Data Types: char | string | cell

### **SupportPackages — Support packages**

'autodetect' (default) | 'none' | string scalar | cell array of character vectors | string array

Support packages to include, specified as one of the following options:

- 'autodetect' (default) — The dependency analysis process detects and includes the required support packages automatically.
- 'none' — No support packages are included. Using this option can cause runtime errors.
- A string scalar, character vector, or cell array of character vectors — Only the specified support packages are included. To list installed support packages or those used by a specific file, see `compiler.codetools.deployableSupportPackages`.

Example: 'SupportPackages', {'Deep Learning Toolbox Converter for TensorFlow Models', 'Deep Learning Toolbox Model for Places365-GoogLeNet Network'}

Data Types: char | string | cell

### **Verbose — Flag to control build verbosity**

'off' (default) | on/off logical value

Flag to control build verbosity, specified as 'on' or 'off', or as numeric or logical 1 (true) or 0 (false). A value of 'on' is equivalent to true, and 'off' is equivalent to false. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- If you set this property to 'on', then the MATLAB command window displays progress information indicating compiler output during the build process.
- If you set this property to 'off', then the command window does not display progress information.

Example: 'Verbose', 'on'

Data Types: logical

## Output Arguments

### **results** — Build results

`compiler.build.Results` object

Build results, returned as a `compiler.build.Results` object. The `Results` object contains:

- Build type, which is 'cppSharedLibrary'
- Paths to the following:
  - `GettingStarted.html`
  - `v2` folder (matlab-data interface only)
  - `LibraryName.dll` (mwArray interface only)
  - `LibraryName.lib` (mwArray interface only)
  - `LibraryName.h` (mwArray interface only)
- A list of included support packages
- Build options, specified as a `CppSharedLibraryOptions` object

## Version History

Introduced in R2021a

### See Also

`compiler.build.CppSharedLibraryOptions`

### Topics

"Generate a C++ MATLAB Data API Shared Library and Build a C++ Application"

"Generate a C++ mwArray API Shared Library and Build a C++ Application"

# compiler.build.CppSharedLibraryOptions

Options for building C++ shared libraries

## Syntax

```
opts = compiler.build.CppSharedLibraryOptions(FunctionFiles)
opts = compiler.build.CppSharedLibraryOptions(FunctionFiles,Name,Value)
```

## Description

`opts = compiler.build.CppSharedLibraryOptions(FunctionFiles)` creates a C++ shared library options object using the MATLAB files `FunctionFiles`. Use the `CppSharedLibraryOptions` object as an input to the `compiler.build.cppSharedLibrary` function.

`opts = compiler.build.CppSharedLibraryOptions(FunctionFiles,Name,Value)` creates a `CppSharedLibraryOptions` object with options specified using one or more name-value arguments. Options include the library name, output directory, and additional files to include.

## Examples

### Create C++ Shared Library Options Object Using Function Files

Create a `CppSharedLibraryOptions` object using file input.

For this example, use the file `magicsquare.m` located in `matlabroot\extern\examples\compiler`.

```
appFile = fullfile(matlabroot,'extern','examples','compiler','magicsquare.m');
opts = compiler.build.CppSharedLibraryOptions(appFile)
```

`opts =`

`CppSharedLibraryOptions` with properties:

```

    Interface: 'matlab-data'
SampleGenerationFiles: {}
    FunctionFiles: {'C:\Program Files\MATLAB\R2022b\extern\examples\compiler\magicsquare'
DebugBuild: off
    LibraryName: 'magicsquare'
    LibraryVersion: '1.0.0.0'
AdditionalFiles: {}
AutoDetectDataFiles: on
SupportPackages: {'autodetect'}
    Verbose: off
    OutputDir: '.\magicsquarecppSharedLibrary'
```

You can modify the property values of an existing `CppSharedLibraryOptions` object using dot notation. For example, enable verbose output.

```
opts.Verbose = 'on'
```

```
opts =
```

```
    CppSharedLibraryOptions with properties:
```

```
        Interface: 'matlab-data'
SampleGenerationFiles: {}
        FunctionFiles: {'C:\Program Files\MATLAB\R2022b\extern\examples\compiler\magicsquare
        DebugBuild: off
        LibraryName: 'magicsquare'
        LibraryVersion: '1.0.0.0'
        AdditionalFiles: {}
AutoDetectDataFiles: on
        SupportPackages: {'autodetect'}
        Verbose: on
        OutputDir: '.\magicsquarecppSharedLibrary'
```

Use the `CppSharedLibraryOptions` object as an input to the `compiler.build.cppSharedLibrary` function to build a C++ shared library.

```
buildResults = compiler.build.cppSharedLibrary(opts);
```

### Customize C++ Shared Library Options Object

Create a `CppSharedLibraryOptions` object and customize it using name-value arguments.

For this example, use the file `magicsquare.m` located in `matlabroot\extern\examples\compiler`. Use name-value arguments to specify the output directory and disable automatic detection of data files.

```
appFile = fullfile(matlabroot,'extern','examples','compiler','magicsquare.m');
opts = compiler.build.CppSharedLibraryOptions(appFile,...
        'OutputDir','D:\Documents\MATLAB\work\MagicSquareLib',...
        'AutoDetectDataFiles','off')
```

```
opts =
```

```
    CppSharedLibraryOptions with properties:
```

```
        Interface: 'matlab-data'
SampleGenerationFiles: {}
        FunctionFiles: {'C:\Program Files\MATLAB\R2022b\extern\examples\compiler\magicsquare
        DebugBuild: off
        LibraryName: 'magicsquare'
        LibraryVersion: '1.0.0.0'
        AdditionalFiles: {}
AutoDetectDataFiles: off
        SupportPackages: {'autodetect'}
        Verbose: off
        OutputDir: 'D:\Documents\MATLAB\work\MagicSquareLib'
```

You can modify the property values of an existing `CppSharedLibraryOptions` object using dot notation. For example, enable verbose output.

```
opts.Verbose = 'on'
```

```
opts =
```



CppSharedLibraryOptions with properties:

```

        Interface: 'matlab-data'
SampleGenerationFiles: {}
        FunctionFiles: {'C:\Program Files\MATLAB\R2022b\extern\examples\compiler\magicsquare'
        DebugBuild: off
        LibraryName: 'magicsquare'
        LibraryVersion: '1.0.0.0'
        AdditionalFiles: {}
AutoDetectDataFiles: off
        SupportPackages: {'autodetect'}
        Verbose: on
        OutputDir: 'D:\Documents\MATLAB\work\MagicSquareLib'

```

Use the CppSharedLibraryOptions object as an input to the compiler.build.cppSharedLibrary function to build a C++ shared library.

```
buildResults = compiler.build.cppSharedLibrary(opts);
```

## Input Arguments

### FunctionFiles — Files implementing MATLAB functions

character vector | string scalar | cell array of character vectors | string array

Files implementing MATLAB functions, specified as a character vector, a string scalar, a string array, or a cell array of character vectors. File paths can be relative to the current working directory or absolute. Files must have a .m extension.

Example: ["myfunc1.m", "myfunc2.m"]

Data Types: char | string | cell

### Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1, ..., NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: 'Verbose', 'on'

### AdditionalFiles — Additional files

character vector | string scalar | cell array of character vectors | string array

Additional files and folders to include in the C++ shared library, specified as a character vector, a string scalar, a string array, or a cell array of character vectors. Paths can be relative to the current working directory or absolute.

Example: 'AdditionalFiles', ["myvars.mat", "data.txt"]

Data Types: char | string | cell

### AutoDetectDataFiles — Flag to automatically include data files

'on' (default) | on/off logical value

Flag to automatically include data files, specified as 'on' or 'off', or as numeric or logical 1 (true) or 0 (false). A value of 'on' is equivalent to true, and 'off' is equivalent to false. Thus, you can

use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- If you set this property to 'on', then data files that you provide as inputs to certain functions (such as `load` and `fopen`) are automatically included in the shared library.
- If you set this property to 'off', then you must add data files to the shared library using the `AdditionalFiles` property.

Example: `'AutoDetectDataFiles','off'`

Data Types: `logical`

### **DebugBuild — Flag to enable debug symbols**

'off' (default) | on/off logical value

Flag to enable debug symbols, specified as 'on' or 'off', or as numeric or logical 1 (`true`) or 0 (`false`). A value of 'on' is equivalent to `true`, and 'off' is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- If you set this property to 'on', then the compiled library contains debug symbols.
- If you set this property to 'off', then the compiled library does not contain debug symbols.

Example: `'DebugBuild','on'`

Data Types: `logical`

### **Interface — Interface API**

'matlab-data' (default) | 'mwstring'

Interface API, specified as one of the following options:

- 'matlab-data' — Generate shared libraries using the MATLAB Data API.
- 'mwstring' — Generate shared libraries using the `mwstring` API.

For more information, see “API Selection for C++ Shared Library”.

Example: `'Interface','mwstring'`

### **LibraryName — Name of C++ shared library**

character vector | string scalar

Name of the C++ shared library, specified as a character vector or a string scalar. The default name of the generated library is the first entry of the `FunctionFiles` argument.

Example: `'LibraryName','mymagic'`

Data Types: `char` | `string`

### **OutputDir — Path to output directory**

character vector | string scalar

Path to the output directory where the build files are saved, specified as a character vector or a string scalar. The path can be relative to the current working directory or absolute.

The default name of the build folder is the library name appended with `cppSharedLibrary`.

Example: `'OutputDir', 'D:\Documents\MATLAB\work\mymagiccppSharedLibrary'`

Data Types: `char | string`

### SampleGenerationFiles — MATLAB sample files

character vector | string scalar | cell array of character vectors | string array

MATLAB sample files used to generate sample C++ library files for functions included within the library, specified as a character vector, a string scalar, a string array, or a cell array of character vectors. Paths can be relative to the current working directory or absolute. Files must have a `.m` extension. For more information and limitations, see “Sample Driver File Creation”.

Example: `'SampleGenerationFiles', ["sample1.m", "sample2.m"]`

Data Types: `char | string | cell`

### SupportPackages — Support packages

'autodetect' (default) | 'none' | string scalar | cell array of character vectors | string array

Support packages to include, specified as one of the following options:

- 'autodetect' (default) — The dependency analysis process detects and includes the required support packages automatically.
- 'none' — No support packages are included. Using this option can cause runtime errors.
- A string scalar, character vector, or cell array of character vectors — Only the specified support packages are included. To list installed support packages or those used by a specific file, see `compiler.codetools.deployableSupportPackages`.

Example: `'SupportPackages', {'Deep Learning Toolbox Converter for TensorFlow Models', 'Deep Learning Toolbox Model for Places365-GoogLeNet Network'}`

Data Types: `char | string | cell`

### Verbose — Flag to control build verbosity

'off' (default) | on/off logical value

Flag to control build verbosity, specified as 'on' or 'off', or as numeric or logical 1 (true) or 0 (false). A value of 'on' is equivalent to `true`, and 'off' is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- If you set this property to 'on', then the MATLAB command window displays progress information indicating compiler output during the build process.
- If you set this property to 'off', then the command window does not display progress information.

Example: `'Verbose', 'on'`

Data Types: `logical`

## Output Arguments

### opts — C++ library build options

CppSharedLibraryOptions object

C++ library build options, returned as a CppSharedLibraryOptions object.

## **Version History**

**Introduced in R2021a**

### **See Also**

`compiler.build.cppSharedLibrary`

# compiler.build.cSharedLibrary

Create C shared library

## Syntax

```
compiler.build.cSharedLibrary(FunctionFiles)
compiler.build.cSharedLibrary(FunctionFiles,Name,Value)
compiler.build.cSharedLibrary(opts)
results = compiler.build.cSharedLibrary( ___ )
```

## Description

`compiler.build.cSharedLibrary(FunctionFiles)` creates a C shared library using the MATLAB files specified by `FunctionFiles`.

`compiler.build.cSharedLibrary(FunctionFiles,Name,Value)` creates a C shared library with options specified using one or more name-value arguments. Options include the library name, output directory, and additional files to include.

`compiler.build.cSharedLibrary(opts)` creates a C shared library with options specified using a `compiler.build.CSharedLibraryOptions` object `opts`. You cannot specify any other options using name-value arguments.

`results = compiler.build.cSharedLibrary( ___ )` returns build information as a `compiler.build.Results` object using any of the input argument combinations in previous syntaxes. The build information consists of the build type, paths to the compiled files, and build options.

## Examples

### Create C Library Using File Input

Create a C shared library using a function file that generates a magic square.

In MATLAB, locate the MATLAB function that you want to deploy as a C shared library. For this example, use the file `magicsquare.m` located in `matlabroot\extern\examples\compiler`.

```
appFile = fullfile(matlabroot,'extern','examples','compiler','magicsquare.m');
```

Build a C library using the `compiler.build.cSharedLibrary` command.

```
compiler.build.cSharedLibrary(appFile);
```

The build function creates the following files within a folder named `magicsquarecSharedLibrary` in your current working directory:

- `GettingStarted.html` — File that contains information on integrating your shared library.
- `includedSupportPackages.txt` — Text file that lists all support files included in the library.
- `magicsquare.c` — C source code file.

- `magicsquare.def` — Module-definition file that provides the linker with module information.
- `magicsquare.dll` — Dynamic-link library file.
- `magicsquare.exports` — Exports file that contains all nonstatic function names.
- `magicsquare.h` — C header file.
- `magicsquare.lib` — Import library file.
- `mccExcludedFiles.log` — Log file that contains a list of any toolbox functions that were not included in the application. For information on non-supported functions, see MATLAB Compiler Limitations.
- `readme.txt` — Readme file that contains information on deployment prerequisites and the list of files to package for deployment.
- `requiredMCRProducts.txt` — Text file that contains product IDs of products required by MATLAB Runtime to run the application.
- `unresolvedSymbols.txt` — Text file that contains information on unresolved symbols.

### Customize C Library

Create a C library and customize it using name-value arguments.

For this example, use the files `flames.m` and `flames.mat` located in `matlabroot\extern\examples\compiler`.

```
appFile = fullfile(matlabroot,'extern','examples','compiler','flames.m');
MATFile = fullfile(matlabroot,'extern','examples','compiler','flames.mat');
```

Build a C library using the `compiler.build.cSharedLibrary` command. Use name-value arguments to specify the library name, add a MAT-file, and enable verbose output.

```
compiler.build.cSharedLibrary(appFile,'LibraryName','MyMagicSquare',...
    'AdditionalFiles',MATFile,...
    'Verbose','on');
```

### Create Multiple Libraries Using Options Object

Create multiple C libraries using a `compiler.build.CSharedLibraryOptions` object.

For this example, use the file `magicsquare.m` located in `matlabroot\extern\examples\compiler`.

```
appFile = fullfile(matlabroot,'extern','examples','compiler','magicsquare.m');
```

Create a `CSharedLibraryOptions` object using `appFile`. Use name-value arguments to specify a common output directory, disable automatic detection of data files, and enable verbose output.

```
opts = compiler.build.CSharedLibraryOptions(appFile,...
    'OutputDir','D:\Documents\MATLAB\work\CLibraryBatch',...
    'AutoDetectDataFiles','off',...
    'Verbose','on')
```

```
opts =
```

```
CSharedLibraryOptions with properties:
```

```
    EmbedArchive: on
    FunctionFiles: {'C:\Program Files\MATLAB\R2022b\extern\examples\compiler\magicsquare.m
```

```

        DebugBuild: off
        LibraryName: 'magicsquare'
        LibraryVersion: '1.0.0.0'
        AdditionalFiles: {}
    AutoDetectDataFiles: off
        SupportPackages: {'autodetect'}
        Verbose: on
        OutputDir: 'D:\Documents\MATLAB\work\CLibraryBatch'

```

Build the C shared library using the `CSharedLibraryOptions` object.

```
compiler.build.cSharedLibrary(opts);
```

To compile using the function file `myMagic2.m` with the same options, use dot notation to modify the `FunctionFiles` argument of the existing `cSharedLibrary` object before running the build function again.

```
opts.FunctionFiles = 'myMagic2.m';
compiler.build.cSharedLibrary(opts);
```

By modifying the `FunctionFiles` argument and recompiling, you can compile multiple libraries using the same options object.

## Get Build Information from C Library

Create a C library and save information about the build type, compiled files, included support packages, and build options to a `compiler.build.Results` object.

Compile using the file `magicsquare.m` located in `matlabroot\extern\examples\compiler`.

```
results = compiler.build.cSharedLibrary('magicsquare.m')
```

```
results =
```

```
    Results with properties:
```

```

        BuildType: 'cSharedLibrary'
        Files: {4×1 cell}
    IncludedSupportPackages: {}
        Options: [1×1 compiler.build.CSharedLibraryOptions]

```

The `Files` property contains the paths to the following files:

- `magicsquare.dll`
- `magicsquare.lib`
- `magicsquare.h`
- `GettingStarted.html`

## Input Arguments

### FunctionFiles — Files implementing MATLAB functions

character vector | string scalar | cell array of character vectors | string array

Files implementing MATLAB functions, specified as a character vector, a string scalar, a string array, or a cell array of character vectors. File paths can be relative to the current working directory or absolute. Files must have a `.m` extension.

Example: ["myfunc1.m", "myfunc2.m"]

Data Types: char | string | cell

### **opts — C library build options**

compiler.build.CSharedLibraryOptions object

C library build options, specified as a compiler.build.CSharedLibraryOptions object.

### **Name-Value Pair Arguments**

Specify optional pairs of arguments as Name1=Value1, . . . , NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: 'Verbose', 'on'

### **AdditionalFiles — Additional files**

character vector | string scalar | cell array of character vectors | string array

Additional files and folders to include in the C shared library, specified as a character vector, a string scalar, a string array, or a cell array of character vectors. Paths can be relative to the current working directory or absolute.

Example: 'AdditionalFiles', ["myvars.mat", "data.txt"]

Data Types: char | string | cell

### **AutoDetectDataFiles — Flag to automatically include data files**

'on' (default) | on/off logical value

Flag to automatically include data files, specified as 'on' or 'off', or as numeric or logical 1 (true) or 0 (false). A value of 'on' is equivalent to true, and 'off' is equivalent to false. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type matlab.lang.OnOffSwitchState.

- If you set this property to 'on', then data files that you provide as inputs to certain functions (such as load and fopen) are automatically included in the shared library.
- If you set this property to 'off', then you must add data files to the shared library using the AdditionalFiles option.

Example: 'AutoDetectDataFiles', 'off'

Data Types: logical

### **DebugBuild — Flag to enable debug symbols**

'off' (default) | on/off logical value

Flag to enable debug symbols, specified as 'on' or 'off', or as numeric or logical 1 (true) or 0 (false). A value of 'on' is equivalent to true, and 'off' is equivalent to false. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type matlab.lang.OnOffSwitchState.

- If you set this property to 'on', then the compiled library contains debug symbols.



- If you set this property to 'off', then the compiled library does not contain debug symbols.

Example: 'DebugBuild', 'on'

Data Types: logical

### **EmbedArchive — Flag to embed deployable archive**

'on' (default) | on/off logical value

Flag to embed the deployable archive, specified as 'on' or 'off', or as numeric or logical 1 (true) or 0 (false). A value of 'on' is equivalent to true, and 'off' is equivalent to false. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- If you set this property to 'on', then the function embeds the deployable archive in the shared library.
- If you set this property to 'off', then the function generates the deployable archive as a separate file.

Example: 'EmbedArchive', 'off'

Data Types: logical

### **LibraryName — Name of C shared library**

character vector | string scalar

Name of the C shared library, specified as a character vector or a string scalar. The default name of the generated library is the first entry of the `FunctionFiles` argument.

Example: 'LibraryName', 'mymagic'

Data Types: char | string

### **OutputDir — Path to output directory**

character vector | string scalar

Path to the output directory where the build files are saved, specified as a character vector or a string scalar. The path can be relative to the current working directory or absolute.

The default name of the build folder is the library name appended with `cSharedLibrary`.

Example: 'OutputDir', 'D:\Documents\MATLAB\work\mymagiccSharedLibrary'

Data Types: char | string

### **SupportPackages — Support packages**

'autodetect' (default) | 'none' | string scalar | cell array of character vectors | string array

Support packages to include, specified as one of the following options:

- 'autodetect' (default) — The dependency analysis process detects and includes the required support packages automatically.
- 'none' — No support packages are included. Using this option can cause runtime errors.
- A string scalar, character vector, or cell array of character vectors — Only the specified support packages are included. To list installed support packages or those used by a specific file, see `compiler.codetools.deployableSupportPackages`.

Example: 'SupportPackages',{ 'Deep Learning Toolbox Converter for TensorFlow Models', 'Deep Learning Toolbox Model for Places365-GoogLeNet Network' }

Data Types: char | string | cell

### **Verbose — Flag to control build verbosity**

'off' (default) | on/off logical value

Flag to control build verbosity, specified as 'on' or 'off', or as numeric or logical 1 (true) or 0 (false). A value of 'on' is equivalent to true, and 'off' is equivalent to false. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- If you set this property to 'on', then the MATLAB command window displays progress information indicating compiler output during the build process.
- If you set this property to 'off', then the command window does not display progress information.

Example: 'Verbose', 'on'

Data Types: logical

## **Output Arguments**

### **results — Build results**

`compiler.build.Results` object

Build results, returned as a `compiler.build.Results` object. The Results object contains:

- Build type, which is 'cSharedLibrary'
- Paths to the following compiled files:
  - *LibraryName.dll*
  - *LibraryName.lib*
  - *LibraryName.h*
  - *GettingStarted.html*
- A list of included support packages
- Build options, specified as a `CSharedLibraryOptions` object

## **Version History**

**Introduced in R2021a**

### **See Also**

`compiler.build.CSharedLibraryOptions`

# compiler.build.CSharedLibraryOptions

Options for building C shared libraries

## Syntax

```
opts = compiler.build.CSharedLibraryOptions(FunctionFiles)
opts = compiler.build.CSharedLibraryOptions(FunctionFiles,Name,Value)
```

## Description

`opts = compiler.build.CSharedLibraryOptions(FunctionFiles)` creates a default C shared library options object using the MATLAB files `FunctionFiles`. Use the `CSharedLibraryOptions` object as an input to the `compiler.build.cSharedLibrary` function.

`opts = compiler.build.CSharedLibraryOptions(FunctionFiles,Name,Value)` creates a `CSharedLibraryOptions` object with options specified using one or more name-value arguments. Options include the library name, output directory, and additional files to include.

## Examples

### Create C Shared Library Options Object Using Function File

Create a `CSharedLibraryOptions` object using file input.

For this example, use the file `magicsquare.m` located in `matlabroot\extern\examples\compiler`.

```
appFile = fullfile(matlabroot,'extern','examples','compiler','magicsquare.m');
opts = compiler.build.CSharedLibraryOptions(appFile)
```

```
opts =
```

```
    CSharedLibraryOptions with properties:
```

```
        EmbedArchive: on
        FunctionFiles: {'C:\Program Files\MATLAB\R2022b\extern\examples\compiler\magicsquare.m'
        DebugBuild: off
        LibraryName: 'magicsquare'
        LibraryVersion: '1.0.0.0'
        AdditionalFiles: {}
        AutoDetectDataFiles: on
        SupportPackages: {'autodetect'}
        Verbose: off
        OutputDir: '.\magicsquarecSharedLibrary'
```

You can modify the property values of an existing `CSharedLibraryOptions` object using dot notation. For example, enable verbose output.

```
opts.Verbose = 'on'
```

```
opts =
```

CSharedLibraryOptions with properties:

```

    EmbedArchive: on
    FunctionFiles: {'C:\Program Files\MATLAB\R2022b\extern\examples\compiler\magicsquare.m
    DebugBuild: off
    LibraryName: 'magicsquare'
    LibraryVersion: '1.0.0.0'
    AdditionalFiles: {}
    AutoDetectDataFiles: on
    SupportPackages: {'autodetect'}
    Verbose: on
    OutputDir: '.\magicsquarecSharedLibrary'

```

Use the CSharedLibraryOptions object as an input to the `compiler.build.cSharedLibrary` function to build a C shared library.

```
buildResults = compiler.build.cSharedLibrary(opts);
```

### Customize C Shared Library Options Object

Create a CSharedLibraryOptions object and customize it using name-value arguments.

For this example, use the file `magicsquare.m` located in `matlabroot\extern\examples\compiler`. Use name-value arguments to specify the output directory and disable automatic detection of data files.

```

appFile = fullfile(matlabroot, 'extern', 'examples', 'compiler', 'magicsquare.m');
opts = compiler.build.CSharedLibraryOptions(appFile, ...
    'OutputDir', 'D:\Documents\MATLAB\work\MagicSquareLib', ...
    'AutoDetectDataFiles', 'off')

```

```
opts =
```

CSharedLibraryOptions with properties:

```

    EmbedArchive: on
    FunctionFiles: {'C:\Program Files\MATLAB\R2022b\extern\examples\compiler\magicsquare.m
    DebugBuild: off
    LibraryName: 'magicsquare'
    LibraryVersion: '1.0.0.0'
    AdditionalFiles: {}
    AutoDetectDataFiles: off
    SupportPackages: {'autodetect'}
    Verbose: off
    OutputDir: 'D:\Documents\MATLAB\work\MagicSquareLib'

```

You can modify the property values of an existing CSharedLibraryOptions object using dot notation. For example, enable verbose output.

```
opts.Verbose = 'on'
```

```
opts =
```

CSharedLibraryOptions with properties:

```

    EmbedArchive: on
    FunctionFiles: {'C:\Program Files\MATLAB\R2022b\extern\examples\compiler\magicsquare.m

```

```

    DebugBuild: off
    LibraryName: 'magicsquare'
    LibraryVersion: '1.0.0.0'
    AdditionalFiles: {}
    AutoDetectDataFiles: off
    SupportPackages: {'autodetect'}
    Verbose: on
    OutputDir: 'D:\Documents\MATLAB\work\MagicSquareLib'

```

Use the `CSharedLibraryOptions` object as an input to the `compiler.build.cSharedLibrary` function to build a C shared library.

```
buildResults = compiler.build.cSharedLibrary(opts);
```

## Input Arguments

### FunctionFiles — Files implementing MATLAB functions

character vector | string scalar | cell array of character vectors | string array

Files implementing MATLAB functions, specified as a character vector, a string scalar, a string array, or a cell array of character vectors. File paths can be relative to the current working directory or absolute. Files must have a `.m` extension.

Example: `["myfunc1.m", "myfunc2.m"]`

Data Types: `char` | `string` | `cell`

### Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose `Name` in quotes.*

Example: `'Verbose', 'on'`

### AdditionalFiles — Additional files

character vector | string scalar | cell array of character vectors | string array

Additional files and folders to include in the C shared library, specified as a character vector, a string scalar, a string array, or a cell array of character vectors. Paths can be relative to the current working directory or absolute.

Example: `'AdditionalFiles', ["myvars.mat", "data.txt"]`

Data Types: `char` | `string` | `cell`

### AutoDetectDataFiles — Flag to automatically include data files

'on' (default) | on/off logical value

Flag to automatically include data files, specified as `'on'` or `'off'`, or as numeric or logical 1 (true) or 0 (false). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- If you set this property to `'on'`, then data files that you provide as inputs to certain functions (such as `load` and `fopen`) are automatically included in the shared library.

- If you set this property to 'off', then you must add data files to the shared library using the `AdditionalFiles` option.

Example: `'AutoDetectDataFiles','off'`

Data Types: `logical`

### **DebugBuild — Flag to enable debug symbols**

'off' (default) | on/off logical value

Flag to enable debug symbols, specified as 'on' or 'off', or as numeric or logical 1 (`true`) or 0 (`false`). A value of 'on' is equivalent to `true`, and 'off' is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- If you set this property to 'on', then the compiled library contains debug symbols.
- If you set this property to 'off', then the compiled library does not contain debug symbols.

Example: `'DebugBuild','on'`

Data Types: `logical`

### **EmbedArchive — Flag to embed deployable archive**

'on' (default) | on/off logical value

Flag to embed the deployable archive, specified as 'on' or 'off', or as numeric or logical 1 (`true`) or 0 (`false`). A value of 'on' is equivalent to `true`, and 'off' is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- If you set this property to 'on', then the function embeds the deployable archive in the shared library.
- If you set this property to 'off', then the function generates the deployable archive as a separate file.

Example: `'EmbedArchive','off'`

Data Types: `logical`

### **LibraryName — Name of C shared library**

character vector | string scalar

Name of the C shared library, specified as a character vector or a string scalar. The default name of the generated library is the first entry of the `FunctionFiles` argument.

Example: `'LibraryName','mymagic'`

Data Types: `char` | `string`

### **OutputDir — Path to output directory**

character vector | string scalar

Path to the output directory where the build files are saved, specified as a character vector or a string scalar. The path can be relative to the current working directory or absolute.

The default name of the build folder is the library name appended with `cSharedLibrary`.

Example: `'OutputDir','D:\Documents\MATLAB\work\mymagiccSharedLibrary'`

Data Types: char | string

### SupportPackages — Support packages

'autodetect' (default) | 'none' | string scalar | cell array of character vectors | string array

Support packages to include, specified as one of the following options:

- 'autodetect' (default) — The dependency analysis process detects and includes the required support packages automatically.
- 'none' — No support packages are included. Using this option can cause runtime errors.
- A string scalar, character vector, or cell array of character vectors — Only the specified support packages are included. To list installed support packages or those used by a specific file, see `compiler.codetools.deployableSupportPackages`.

Example: 'SupportPackages', {'Deep Learning Toolbox Converter for TensorFlow Models', 'Deep Learning Toolbox Model for Places365-GoogLeNet Network'}

Data Types: char | string | cell

### Verbose — Flag to control build verbosity

'off' (default) | on/off logical value

Flag to control build verbosity, specified as 'on' or 'off', or as numeric or logical 1 (true) or 0 (false). A value of 'on' is equivalent to true, and 'off' is equivalent to false. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- If you set this property to 'on', then the MATLAB command window displays progress information indicating compiler output during the build process.
- If you set this property to 'off', then the command window does not display progress information.

Example: 'Verbose', 'on'

Data Types: logical

## Output Arguments

### opts — C library build options

CSharedLibraryOptions object

C library build options, returned as a CSharedLibraryOptions object.

## Version History

Introduced in R2021a

### See Also

`compiler.build.cSharedLibrary`

## <library>Initialize[WithHandlers]

Initialize MATLAB Runtime instance associated with *library*

### Syntax

```
bool libraryInitialize()
bool libraryInitializeWithHandlers(mclOutputHandlerFcn error_handler,
mclOutputHandlerFcn print_handler)
```

### Description

`bool libraryInitialize()` creates a MATLAB Runtime instance associated with the generated C/C++ shared library *library* using the default print and error handlers.

Call this function in your C/C++ application after calling `mclInitializeApplication` and before calling any of the compiled functions exported by the library.

`bool libraryInitializeWithHandlers(mclOutputHandlerFcn error_handler, mclOutputHandlerFcn print_handler)` creates a MATLAB Runtime instance associated with *library* and allows you to specify how to handle error messages and printed text. The functions passed to `libraryInitializeWithHandlers` are installed in the MATLAB Runtime instance and called whenever error text or regular text outputs.

### Examples

#### Initialize Library

Initialize the MATLAB Runtime instance associated with a C/C++ library named `libtriangle`.

Call the initialization function for the C library `libtriangle` in the main function of your C application code.

```
if (!libtriangleInitialize())
{
    fprintf(stderr,
            "An error occurred while initializing: \n %s ",
            mclGetLastErrorMessage());
    return -2;
}
```

If initialization fails, an error message is printed to standard error.

#### Initialize Library With Handlers

By default, applications and shared libraries generated using MATLAB Compiler SDK send printed output to standard output and error messages to standard error. To change this behavior, write your own error and print handlers and pass them to `libraryInitializeWithHandlers`. To use a default handler, pass in `mclDefaultPrintHandler` or `mclDefaultPrintHandler`.



For a complete example, see the files located in *matlabroot*\extern\examples\compilersdk\c\_cpp\catcherror.

Write a custom print handler function in your C/C++ application. This example code is written in C and uses two helper functions, *StartMatlabOutput* and *EndMatlabOutput*, to wrap a banner around the printed output.

```
static int PrintHandler(const char *s)
{
    /* Declare and initialize all variables */
    int len = 0;
    const char * prefix = "* ";
    int written = 0;

    if (s == NULL)
    {
        return 0;
    }

    len = strlen(s);

    /* DISP adds two carriage returns. Suppress the last one. */
    if (len >= 2 && s[len-1] == '\n' && s[len-2] == '\n')
    {
        len = len-1;
    }

    if (emitPrefix)
    {
        fwrite(prefix, sizeof(char), strlen(prefix), stdout);
    }

    written = fwrite(s, sizeof(char), len, stdout);

    if (s[len-1] == '\n')
    {
        emitPrefix = true;
    }
    else
    {
        emitPrefix = false;
    }

    return written;
}

static void StartMatlabOutput()
{
    const char *startBanner = "***** Start MATLAB output *****\n";
    emitPrefix = false;
    PrintHandler(startBanner);
}

static void EndMatlabOutput()
{
    const char *endBanner = "***** End MATLAB output *****\n";
    emitPrefix = false;
}
```

```
    PrintHandler(endBanner);
}
```

Write a custom error handler function.

```
static char LastError[2048];

static int ErrorHandler(const char *s)
{
    int len = 0;
    len = strlen(s);
    LastError[0] = '\0';
    strcpy(LastError, s);
    return len+1;
}
```

In your main function, after initializing variables, pass the print and error handlers to the `libraryInitializeWithHandlers` function.

```
if (!libcatcherrorInitializeWithHandlers(ErrorHandler, PrintHandler))
{
    fprintf(stderr, "Could not initialize the library.\n");
    return -2;
}
else
{
    /* Call the library function. */
    if (mlfRealacos(1, &out, in1))
    {
        /* Display the return value of the library function */
        printf("realacos(%6.4f) = \n", in1val);
        StartMatlabOutput();
        mlfReveal(out);
        EndMatlabOutput();
    }
    else
    {
        printf("Disaster! An unexpected error occurred.\n");
        printf("Error:\n%s\n", LastError);
    }

    /* Call the library termination routine */
    libcatcherrorTerminate();

    /* Free the memory allocated for the input variables. */
    mxDestroyArray(in1);
    in1 = 0;
    mxDestroyArray(in2);
    in2 = 0;
}
```

## Input Arguments

### library — Library name

Library name, specified as part of the function name. The library must be a C/C++ shared library generated by MATLAB Compiler SDK.

Example: `libmatrix`

**error\_handler — Error handler function**

custom function | `mclDefaultErrorHandler`

Error handler function, specified as a custom user-written function or `mclDefaultErrorHandler`. The function must take a string and return the number of characters printed. For more details, see “Print and Error Handling Functions” on page 2-19.

Example: `myErrorHandler`

**print\_handler — Print handler function**

custom function | `mclDefaultPrintHandler`

Print handler function, specified as a custom user-written function or `mclDefaultPrintHandler`. The function must take a string and return the number of characters printed. If the string sent to the standard error output stream does not end with a carriage return, the function must add one. For more details, see “Print and Error Handling Functions” on page 2-19.

Example: `myPrintHandler`

## Output Arguments

**bool — Initialization result**

true | false

Initialization result, returned as a boolean value. Result indicates whether or not `mcli` initialization was successful. If the function returns `false`, calling any compiled functions results in unpredictable behavior.

## Version History

Introduced in R2009a

## See Also

<library>Terminate | `mclInitializeApplication` | `mclTerminateApplication`

### Topics

“Library Initialization and Termination Functions” on page 2-18

“Print and Error Handling Functions” on page 2-19

“Call a C Shared Library” on page 2-2

## mcc

Compile MATLAB functions for deployment

### Syntax

mcc

```
mcc options mfilename1 mfilename2 ... mfilenameN
mcc('options','mfilename')
```

```
mcc -l options mfilename1 mfilename2 ... mfilenameN
```

```
mcc -W cpplib:library_name[, {all|legacy|generic}] options mfilename1
mfilename2 ... mfilenameN
```

```
mcc -W com:component_name,className options
class{className:mfilename1,mfilename2,...,mfilenameN}
```

```
mcc -W
dotnet:assembly_name,api=api_type,className,framework_version,security,remote
_type -T link:lib options mfilename1 mfilename2 ... mfilenameN
```

```
mcc -W
dotnet:assembly_name,api=api_type,className,framework_version,security,remote
_type -T link:lib options
class{className:mfilename1,mfilename2,...,mfilenameN}
```

```
mcc -W java:packageName,className options mfilename1 mfilename2 ...
mfilenameN
```

```
mcc -W java:packageName,className options
class{className:mfilename1,mfilename2,...,mfilenameN}
```

```
mcc -W python:packageName -T link:lib options mfilename1 mfilename2 ...
mfilenameN
```

```
mcc -W CTF:archive_name -U options mfilename1 mfilename2 ... mfilenameN
```

```
mcc -W mpsxl:addin_name,className,version input_marshaling_flags
output_marshaling_flags -T link:lib options mfilename1 mfilename2 ...
mfilenameN
```

### Description

#### Other mcc Topics

This topic describes mcc options for use with MATLAB Compiler SDK. For information on compiling:

- standalone applications, Excel add-ins, or Hadoop® jobs, see `mcc` (MATLAB Compiler)
- MATLAB Production Server deployable archives or Excel add-ins for MATLAB Production Server, see `mcc`

## General Usage

`mcc options mfilename1 mfilename2 ... mfilenameN` compiles the functions as specified by the options. The options used depend on the intended results of the compilation.

`mcc('options', 'mfilename')` compiles the function as specified by the options. Specify filenames, options, and option parameters in single quotes.

## C Shared Library

`mcc -l options mfilename1 mfilename2 ... mfilenameN` compiles the listed functions into a C shared library and generates C wrapper code for integration with other applications.

This syntax is equivalent to `-W lib:libname -T link:lib`.

## C++ Shared Library

`mcc -W cpplib:library_name[, {all|legacy|generic}] options mfilename1 mfilename2 ... mfilenameN` compiles the listed functions into a C++ shared library and generates C++ wrapper code for integration with other applications.

- *library\_name* — Specifies the name of the shared library.
- *all*— Generates shared libraries using both the `mwArray` API and the generic interface that uses the MATLAB Data API. This is the default.
- *legacy*— Generates shared libraries using the `mwArrayAPI`.
- *generic*— Generates shared libraries using the MATLAB Data API.

## COM Component

`mcc -W com:component_name,className options class{className:mfilename1,mfilename2,...,mfilenameN}` compiles the listed functions into a generic Microsoft COM component.

---

**Note** File names listed in the `class{_____}` argument must be separated by commas.

You can include multiple class specifiers by adding additional `class{ ___ }` arguments.

---

- *component\_name* — Specifies the name of the COM component.
- *className* — Specifies the name of the class.

## .NET Assembly

`mcc -W dotnet:assembly_name,api=api_type,className,framework_version,security,remote_type -T link:lib options mfilename1 mfilename2 ... mfilenameN` creates a .NET assembly with a single class from the specified files.

- *assembly\_name* — Specifies the name of the assembly preceded by an optional namespace, which is a period-separated list, such as `companyname.groupname.component`.
- *api\_type* — Specifies the API type of the assembly. Values are `matlab-data` and `mwarray`.
- *className* — Specifies the name of the .NET class to be created.

- *framework\_version* — Specifies the version of the Microsoft .NET Framework you want to use to compile the assembly. Specify either:
  - 0.0 — Use the latest supported version on the target machine.
  - *version\_major.version\_minor* — Use a specific version of the framework.

Features are often version-specific. Consult the documentation for the feature you are implementing to get the Microsoft .NET Framework version requirements.

- *security* — Specifies whether the assembly to be created is a private assembly or a shared assembly.
  - To create a private assembly, specify `Private`.
  - To create a shared assembly, specify the full path to the encryption key file used to sign the assembly.
- *remote\_type* — Specifies the remoting type of the assembly. Values are `remote` and `local`.

```
mcc -W
dotnet:assembly_name,api=api_type,className,framework_version,security,remote
_type -T link:lib options
class{className:mfilename1,mfilename2,...,mfilenameN} creates a .NET assembly with
multiple classes from the specified files.
```

---

**Note** File names listed in the `class{_____}` argument must be separated by commas.

You can include multiple class specifiers by adding additional `class{_____}` arguments.

---

### Java Package

```
mcc -W java:packageName,className options mfilename1 mfilename2 ...
mfilenameN creates a Java package from the specified files.
```

- *packageName* — Specifies the name of the Java package preceded by an optional namespace, which is a period-separated list, such as `companyname.groupname.component`.
- *className* — Specifies the name of the class to be created. If you do not specify the class name, `mcc` uses the last item in *packageName*.

```
mcc -W java:packageName,className options
class{className:mfilename1,mfilename2,...,mfilenameN} creates a Java package with
multiple classes from the specified files.
```

---

**Note** File names listed in the `class{_____}` argument must be separated by commas.

- *packageName* — Specifies the name of the Java package preceded by an optional namespace, which is a period-separated list, such as `companyname.groupname.component`.
- *className* — Specifies the name of the class to be created. If you do not specify the class name, `mcc` uses the last item in *packageName*.

---

**Note** You can include multiple class specifiers by adding additional `class{_____}` arguments.

---

### Python Package

`mcc -W python:packageName -T link:lib options mfilename1 mfilename2 ... mfilenameN` creates a Python package from the specified files.

- *packageName* — Specifies the name of the Python package preceded by an optional namespace, which is a period-separated list, such as `companyname.groupname.component`.

### Deployable Archive for MATLAB Production Server

`mcc -W CTF:archive_name -U options mfilename1 mfilename2 ... mfilenameN` instructs the compiler to create a deployable archive (.ctf file) for use with a MATLAB Production Server instance.

The syntax also creates the server-side deployable archive (.ctf file) for Microsoft Excel add-ins.

### Excel Add-In for MATLAB Production Server

`mcc -W mpsxl:addin_name,className,version input_marshall_flags output_marshall_flags -T link:lib options mfilename1 mfilename2 ... mfilenameN` creates a client-side Microsoft Excel add-in from the specified files that can be used to send requests to MATLAB Production Server from Excel. Creating the client-side add-in *must* be preceded by creating a server-side deployable archive (.ctf file) from the specified files. A purely client side add-in is not viable.

- *addin\_name* — Specifies the name of the add-in.
- *className* — Specifies the name of the class to be created. If you do not specify the class name, `mcc` uses the *addin\_name* as the default.
- *version* — Specifies the version of the add-in specified as *major.minor*.
  - *major* — Specifies the major version number. If you do not specify a version number, `mcc` uses the latest version.
  - *minor* — Specifies the minor version number. If you do not specify a version number, `mcc` uses the latest version.
- *input\_marshall\_flags* — Specifies options for how data is marshaled between Microsoft Excel and MATLAB.
  - `-replaceBlankWithNaN` — Specifies that a blank in Microsoft Excel is marshaled into NaN in MATLAB. If you do not specify this flag, blanks are marshaled into 0.
  - `-convertDateToString` — Specifies that dates in Microsoft Excel are marshaled into MATLAB character vectors. If you do not specify this flag, dates are marshaled into MATLAB doubles.
- *output\_marshall\_flags* — Specifies options for how data is marshaled between MATLAB and Microsoft Excel.
  - `-replaceNaNWithZero` — Specifies that NaN in MATLAB is marshaled into a 0 in Microsoft Excel. If you do not specify this flag, NaN is marshaled into #QNAN in Visual Basic.
  - `-convertNumericToDate` — Specifies that MATLAB numeric values are marshaled into Microsoft Excel dates. If you do not specify this flag, Microsoft Excel does not receive dates as output.

## Examples

### Create a standalone application and include MATLAB preferences

```
mcc -m helloWorld.m -a C:\Users\someuser\AppData\Roaming\MathWorks\MATLAB\R2022b\matlab.mlsetting
```

### Create a C shared library

```
mcc -l mymagic.m
```

### Create a C shared library with a system-level file version number (Windows only)

Create a C shared library in Windows with version number 4.3.1.7.

```
mcc -W 'lib:myCSharedLib,version=4.3.1.7' mymagic.m
```

### Create a C++ shared library

Use the mxArray API

```
mcc -W 'cpplib:mymagic,legacy' mymagic.m
```

Use the MATLAB Data API

```
mcc -W 'cpplib:mymagic,generic' mymagic.m
```

Use both the mxArray API and the MATLAB Data API

```
mcc -W 'cpplib:mymagic,all' mymagic.m
```

OR

```
mcc -W 'cpplib:mymagic' mymagic.m
```

### Create a C++ shared library with a system-level file version number (Windows only)

Create a C++ shared library in Windows with version number 3.7.1.5.

```
mcc -W 'cpplib:mymagic,all,version=3.7.1.5' mymagic.m
```

### Create a COM component

Create a COM component in Windows with version number 7.10.1.3.

```
mcc -W 'com:myCOMComponent,myClass,version=7.10.1.3' class{myClass:mymagic.m}
```

### Create a Java package containing multiple classes

```
mcc -W 'java:myMatrix,add' class{add:add.m} class{sub:minus.m}
```

### Create a Python package

```
mcc -W python:myMagic magic.m
```

### Create a deployable archive for MATLAB Production Server

```
mcc -W CTF:myDeployableArchive -U mymagic.m
```



## Create an Excel add-in for MATLAB Production Server

```
mcc -W 'mpsxl:myDeployableArchive,myExcelClass,version=1.0' mymagic.m
```

## Input Arguments

### **mfilename1 mfilename2 ... mfilenameN — Files to be compiled**

list of file names

One or more files to be compiled, specified as a space-separated list of file names. The first file is used as the entry point for the compiled artifact.

### **class{className:mfilename1,mfilename2,...,mfilenameN} — Files to be included in a class**

list of file names

One or more files to be included in the class *className*, specified as a comma-separated list of file names. You can repeat this argument to include multiple class specifiers. The argument applies only to the COM component, Java package, and .NET assembly targets.

### **options — Options for customizing the output**

```
-a | -b | -B | -c | -C | -d | -f | -g | -G | -I | -j | -k | -K | -m | -M | -n | -N | -o | -p | -r | -R | -s | -S | -T | -u | -U | -v | -w | -W | -X | -Y | -Z
```

Options for customizing the output, specified as a list of character vectors or string scalars.

- **-a**

Add files to the deployable archive using `-a filepath` to specify the files to be added. Multiple `-a` options are permitted.

Also, add MATLAB preferences to a deployed application using `-a path\to\mymatlab.mlsettings` to specify the preferences to be added.

If a file name is specified with `-a`, the compiler looks for these files on the MATLAB path, so specifying the full path name is optional. These files are not passed to `mbuild`, so you can include files such as data files. For more details, see “Access Files in Packaged Applications”.

If a folder name is specified with the `-a` option, the entire contents of that folder are added recursively to the deployable archive. For example,

```
mcc -m hello.m -a ./testdir
```

specifies that all files in `testdir`, as well as all files in its subfolders, are added to the deployable archive. The folder subtree in `testdir` is preserved in the deployable archive.

If the filename includes a wildcard pattern, only the files in the folder that match the pattern are added to the deployable archive and subfolders of the given path are not processed recursively. For example,

```
mcc -m hello.m -a ./testdir/*
```

specifies that all files in `./testdir` are added to the deployable archive and subfolders under `./testdir` are not processed recursively.

```
mcc -m hello.m -a ./testdir/*.m
```

specifies that all files with the extension `.m` under `./testdir` are added to the deployable archive and subfolders of `./testdir` are not processed recursively.

---

**Note** `*` is the only supported wildcard.

---

When you add files to the archive using `-a` that do not appear on the MATLAB path at the time of compilation, a path entry is added to the application's run-time path so that they appear on the path when the deployed code executes.

When you use the `-a` option to specify a full path to a resource, the basic path is preserved, with some modifications, but relative to a subdirectory of the runtime cache directory, not to the user's local folder. The cache directory is created from the deployable archive the first time the application is executed. You can use the `isdeployed` function to determine whether the application is being run in deployed mode, and adjust the path accordingly. The `-a` option also creates a `.auth` file for authorization purposes.

---

**Caution** If you use the `-a` flag to include a file that is not on the MATLAB path, the folder containing the file is added to the MATLAB dependency analysis path. As a result, other files from that folder might be included in the compiled application.

---



---

**Note** If you use the `-a` flag to include custom Java classes, standalone applications work without any need to change the `classpath` as long as the Java class is not a member of a package. The same applies for JAR files. However, if the class being added is a member of a package, the MATLAB code needs to make an appropriate call to `javaaddpath` to update the `classpath` with the parent folder of the package.

---

- **-A**

Append supported platforms to those detected automatically by the compiler. Use

`-A arch`

to add the platform designated by `arch` to the list of compatible platforms. Valid platforms are `win64`, `maci64`, `glnxa64`, and `all`. The `-A` option only applies to Python, C/C++ using the MATLAB data array API, and Java targets.

Running the component on an incompatible platform will result in an unsupported platform error message and list compatible platforms.

- **-b**

Generate a Visual Basic file (`.bas`) containing the Microsoft Excel Formula Function interface to the COM object generated by MATLAB Compiler. When imported into the workbook Visual Basic code, this code allows the MATLAB function to be seen as a cell formula function.

- **-B**

Replace the file on the `mcc` command line with the contents of the specified file. Use

`-B filename[:<a1>,<a2>,...,<an>]`

The bundle `filename` should contain only `mcc` command-line options and corresponding arguments and/or other file names. The file might contain other `-B` options. A bundle can include

replacement parameters for compiler options that accept names and version numbers. See “Using Bundles to Build MATLAB Code”.

- **-c**

When used in conjunction with the `-l` option, suppresses compiling and linking of the generated C wrapper code. The `-c` option cannot be used independently of the `-l` option.

- **-C**

Do not embed the deployable archive in binaries.

---

**Note** The `-C` flag is ignored for Java libraries.

---

- **-d**

Place output in a specified folder. Use

`-d outFolder`

to direct the generated files to *outFolder*. The specified folder must already exist.

- **-e**

Use `-e` in place of the `-m` option to generate a standalone Windows application that does not open a Windows command prompt on execution. `-e` is equivalent to `-W WinMain -T link:exe`.

This option works only on Windows operating systems.

- **-f**

Override the default options file with the specified options file. It specifically applies to the C/C++ shared libraries, COM, and Excel targets. Use

`-f filename`

to specify *filename* as the options file when calling `mbuild`. This option lets you use different ANSI compilers for different invocations of the compiler. This option is a direct pass-through to `mbuild`.

- **-g, -G**

Include debugging symbol information for the C/C++ code generated by MATLAB Compiler SDK. It also causes `mbuild` to pass appropriate debugging flags to the system C/C++ compiler. The debug option lets you backtrace up to the point where you can identify if the failure occurred in the initialization of MATLAB Runtime, the function call, or the termination routine. This option does not let you debug your MATLAB files with a C/C++ debugger.

- **-h**

Specify a custom help text file. This option applies to standalone applications, C/C++ shared libraries, COM, and Excel targets. Use

`-f filename`

to include *filename* as the custom help text file. Display help file contents by calling the application at the command line with the `-?` or `/?` argument.

- **-I**

Add a new folder path to the list of included folders. Each `-I` option appends the folder to the end of the list of paths to search. For example,

```
-I <directory1> -I <directory2>
```

sets up the search path so that `directory1` is searched first for MATLAB files, followed by `directory2`. This option is important for standalone compilation where the MATLAB path is not available.

If used in conjunction with the `-N` option, the `-I` option adds the folder to the compilation path in the same position where it appeared in the MATLAB path rather than at the head of the path.

- **-j**

Automatically convert all `.m` files to P-files before packaging. The `-j` option generates a P-code file with a `.p` extension for each `.m` file included in the `mcc` command.

P-code files are an obfuscated, execute-only form of MATLAB code. For more details, see `pcode`.

- **-k**

Specify an AES encryption key and a MEX-file loader interface to retrieve the decryption key at runtime.

Use

```
-k "file=<key_file_path>;loader=<mex_file_path>"
```

to specify paths to the key file and MEX-file.

The key file must be in one of the following supported formats:

- Binary 256-bit AES key, with a 32 byte file size
- Hex encoded AES key, with a 64 character file size

The loader MEX-file must be an interface with the following arguments:

- `prhs[0]` — Input, char array specified as the static value `'getKey'`
- `prhs[1]` — Input, char array specified as the CTF component UUID
- `plhs[0]` — Output, 32 byte UINT8 numeric array or 64 byte HEX encoded char array

If you do not specify any arguments after `-k`, `mcc` generates a 256-bit AES key and a loader MEX-file that can be used for demonstration purposes.

- **-K**

Direct `mcc` to not delete output files if the compilation ends prematurely due to error.

The default behavior of `mcc` is to dispose of any partial output if the command fails to execute successfully.

- **-l**

Generate a function library. `-l` is equivalent to `-W lib -T link:lib`. You cannot use this option in a `deploytool` app.

- **-m**

Generate a standalone application. `-m` is equivalent to `-W main -T link:exe`. You cannot use this option in a `deploytool` app.

- **-M**

Define compile-time options. Use

`-M string`

to pass `string` directly to `mbuild`. This option provides a useful mechanism for defining compile-time options, for example, `-M "-Dmacro=value"`.

---

**Note** Multiple `-M` options do not accumulate; only the rightmost `-M` option is used.

---

To pass options such as `/bigobj`, delineate the string according to your platform.

Platform	Syntax
MATLAB	<code>-M 'COMPFLAGS=\$COMPFLAGS /bigobj'</code>
Windows command prompt	<code>-M COMPFLAGS="\$COMPFLAGS /bigobj"</code>
Linux and macOS command line	<code>-M CFLAGS='\$CFLAGS /bigobj'</code>

- **-n**

The `-n` option automatically identifies numeric command line inputs and treats them as MATLAB doubles.

- **-N**

Passing `-N` clears the path of all folders except the following core folders (this list is subject to change over time):

- `matlabroot\toolbox\matlab`
- `matlabroot\toolbox\local`
- `matlabroot\toolbox\compiler`
- `matlabroot\toolbox\shared\bigdata`

Passing `-N` also retains all subfolders in this list that appear on the MATLAB path at compile time. Including `-N` on the command line lets you replace folders from the original path, while retaining the relative ordering of the included folders. All subfolders of the included folders that appear on the original path are also included. In addition, the `-N` option retains all folders that you included on the path that are not under `matlabroot\toolbox`.

When using the `-N` option, use the `-I` option to force inclusion of a folder, which is placed at the head of the compilation path. Use the `-p` option to conditionally include folders and their subfolders; if they are present in the MATLAB path, they appear in the compilation path in the same order.

- **-o**

Specify the name of the final executable (standalone applications only). Use

`-o outputfile`

to name the final executable output of MATLAB Compiler. A suitable platform-dependent extension is added to the specified name (for example, `.exe` for Windows standalone applications).

- **-p**

Use in conjunction with the option `-N` to add specific folders and subfolders under `matlabroot\toolbox` to the compilation MATLAB path. The files are added in the same order in which they appear in the MATLAB path. Use the syntax

```
-N -p directory
```

where `directory` is the folder to be included. If `directory` is not an absolute path, it is assumed to be under the current working folder.

- If a folder is included with `-p` that is on the original MATLAB path, the folder and all its subfolders that appear on the original path are added to the compilation path in the same order.
- If a folder is included with `-p` that is not on the original MATLAB path, that folder is ignored. (You can use `-I` to force its inclusion.)

- **-r**

Embed resource icon in binary. The syntax is as follows:

```
-r 'path/to/my_icon.ico'
```

- **-R**

Provide MATLAB Runtime options that are passed to the application at initialization time.

---

**Note** This option is relevant only when building standalone applications or Excel add-ins using MATLAB Compiler. If you specify the `-R` option for libraries created from MATLAB Compiler SDK, `mcc` still compiles and generates the results, but the `-R` option doesn't apply to these libraries and does not do anything.

---

The syntax is as follows:

```
-R option
```

Option	Description	Target
'-logfile, filename'	Specify a log file name. The file is created in the application folder at runtime and contains information about MATLAB Runtime initialization and all text piped to the command window. Option must be in single quotes. Use double quotes when executing the command from a Windows Command Prompt.	MATLAB Compiler
-nodisplay	Suppress the MATLAB <code>nodisplay</code> run-time warning. On Linux, open MATLAB Runtime without display functionality.	MATLAB Compiler
-nojvm	Do not use the Java Virtual Machine (JVM).	MATLAB Compiler

Option	Description	Target
' - startmsg ,message '	Customizable user message displayed at initialization time. For more details, see "Display MATLAB Runtime Initialization Messages".	MATLAB Compiler Standalone Applications
' - complete msg ,message '	Customizable user message displayed when initialization is complete. For more details, see "Display MATLAB Runtime Initialization Messages".	MATLAB Compiler Standalone Applications
- singleCompThread	Limit MATLAB to a single computational thread.	MATLAB Compiler
- software opengl	Use Mesa Software OpenGL for rendering.	MATLAB Compiler

**Caution** When running on macOS, if you use `-nodisplay` as one of the options included in `mclInitializeApplication`, then the call to `mclInitializeApplication` must occur before calling `mclRunMain`.

- **-s**

Obfuscate folder structures and file names in the deployable archive (.ctf file) from the end user. Optionally encrypt additional file types.

The `-s` option directs `mcc` to place user code and data contained in `.m`, `.p`, `v7.3 .mat`, and MEX files into a user package within the CTF. During runtime, MATLAB code and data is decrypted and loaded directly from the user package rather than extracted to the file system. MEX files are temporarily extracted from the user package before being loaded.

To manually include additional file types in the user package, add each file type in a separate extension tag to the file `matlabroot/toolbox/compiler/advanced_package_supported_files.xml`.

The following is not supported:

- `ver` function
- Out-of-process MATLAB Runtime ( C++ shared library for MATLAB Data Array)
- Out-of-process MEX file execution (`mexhost`, `feval`, `matlab.mex.MexHost`)

- **-S**

Create a single MATLAB Runtime instance that is shared across all class instances.

The standard behavior for the MATLAB Runtime is that every instance of a class gets its own MATLAB Runtime context. The context includes a global MATLAB workspace for variables, such as the path, and a base workspace for each function in the class. If multiple instances of a class are created, each instance gets an independent context. This ensures that changes made to the global or base workspace in one instance of the class does not affect other instances of the same class.

In a singleton MATLAB Runtime, all instances of a class share the context. If multiple instances of a class are created, they use the context created by the first instance which saves startup time and

some resources. However, any changes made to the global workspace or the base workspace by one instance impacts all class instances. For example, if `instance1` creates a global variable `A` in a singleton MATLAB Runtime, then `instance2` can use variable `A`.

Singleton MATLAB Runtime is only supported by the following products on these specific targets:

Target supported by Singleton MATLAB Runtime	Create a Singleton MATLAB Runtime by....
Excel add-in	Default behavior for target is singleton MATLAB Runtime. You do not need to perform other steps.
.NET assembly	Default behavior for target is singleton MATLAB Runtime. You do not need to perform other steps.
COM component	<ul style="list-style-type: none"> <li>Using the Library Compiler app, click <b>Settings</b> and add <code>-S</code> to the <b>Additional parameters passed to MCC</b> field.</li> <li>Using <code>mcc</code>, pass the <code>-S</code> flag.</li> </ul>
Java package	

- **-T**

Specify the output target phase and type. If not specified, `mcc` uses the default type for the target specified by the `-W` option.

Use the syntax `-T target` to define the output type.

Target	Description
<code>compile:exe</code>	Generate a C/C++ wrapper file, and compile C/C++ files to an object form suitable for linking into a standalone application.
<code>compile:lib</code>	Generate a C/C++ wrapper file, and compile C/C++ files to an object form suitable for linking into a shared library or DLL.
<code>link:exe</code>	Same as <code>compile:exe</code> and also link object files into a standalone application.
<code>link:lib</code>	Same as <code>compile:lib</code> and also link object files into a shared library or DLL.

- **-u**

Register COM component for the current user only on the development machine. The argument applies only to the generic COM component and Microsoft Excel add-in targets.

- **-U**

Build deployable archive (`.ctf` file) for MATLAB Production Server.

- **-v**

Display verbose output. Output displays the compilation steps, including:

- MATLAB Compiler version number
- Source file names as they are processed
- Names of the generated output files as they are created



- Invocation of `mbuild`

The `-v` option passes the `-v` option to `mbuild` and displays information about `mbuild`.

- **-w**

Display warning messages. Use the syntax

```
-w option [:<msg>]
```

to control the display of warnings.

Syntax	Description
<code>-w list</code>	List the compile-time warnings that have abbreviated identifiers, together with their status.
<code>-w enable</code>	Enable all compile-time warnings.
<code>-w disable[:&lt;string&gt;]</code>	Disable specific compile-time warnings associated with <code>&lt;string&gt;</code> . Omit the optional <code>&lt;string&gt;</code> to apply the <code>disable</code> action to all compile-time warnings.
<code>-w enable[:&lt;string&gt;]</code>	Enable specific compile-time warnings associated with <code>&lt;string&gt;</code> . Omit the optional <code>&lt;string&gt;</code> to apply the <code>enable</code> action to all compile-time warnings.
<code>-w error[:&lt;string&gt;]</code>	Treat specific compile-time and runtime warnings associated with <code>&lt;string&gt;</code> as an error. Omit the optional <code>&lt;string&gt;</code> to apply the <code>error</code> action to all compile-time and runtime warnings.
<code>-w off[:&lt;string&gt;]</code>	Turn off warnings for specific error messages defined by <code>&lt;string&gt;</code> . Omit the optional <code>&lt;string&gt;</code> to apply the <code>off</code> action to all runtime warnings.
<code>-w on[:&lt;string&gt;]</code>	Turn on runtime warnings associated with <code>&lt;string&gt;</code> . Omit the optional <code>&lt;string&gt;</code> to apply the <code>on</code> action to all runtime warnings. This option is enabled by default.

You can also turn warnings on or off in your MATLAB code.

For example, to turn off warnings for deployed applications (specified using `isdeployed`) in `startup.m`, you write:

```
if isdeployed
    warning off
end
```

To turn on warnings for deployed applications, you write:

```
if isdeployed
    warning on
end
```

You can also specify multiple `-w` options.

For example, if you want to disable all warnings except `repeated_file`, you write:

```
-w disable -w enable:repeated_file
```

When you specify multiple `-w` options, they are processed from left to right.

- **-W**

Control the generation of function wrappers. Use the syntax

`-W type`

to control the generation of function wrappers for a collection of MATLAB files generated by the compiler. You provide a list of functions, and the compiler generates the wrapper functions and any appropriate global variable definitions.

Target	Syntax
C Shared Library	<code>-W 'lib:libName'</code>
C++ Shared Library	<code>-W 'cpplib:libName[,{all legacy generic}]'</code>
COM Component	<code>-W 'com:comName,className'</code>
.NET Assembly	<code>-W 'dotnet:assemblyName,api={matlab-data mwarray},className,frameworkVersion,security,{remote local}'</code>
Java Package	<code>-W 'java:packageName,className'</code>
Python Package	<code>-W 'python:packageName,className'</code>
MATLAB Production Server Deployable Archive	<code>-W 'CTF:archiveName'</code>
MATLAB Production Server Excel Add-In	<code>-W 'mpsx:addinName,className,version'</code>

---

**Note** Replace single quotes with double when executing the command from a Windows Command Prompt.

- **-X**

Use `-X` to ignore data files read by common MATLAB file I/O functions during dependency analysis. For more information, see “Dependency Analysis Using MATLAB Compiler”. For examples on how to use the `-X` option, see `%#exclude`.

- **-Y**

Use

`-Y license.lic`

to override the default license file with the specified argument.

---

**Note** The `-Y` flag works only with the command-line mode.

```
>>!mcc -m foo.m -Y license.lic
```

- **-Z**

Use

*-Z option*

to specify the method of adding support packages to the deployable archive.

Syntax	Description
<code>-Z 'autodetect'</code>	The dependency analysis process detects and includes the required support packages automatically. This is the default behavior of <code>mcc</code> .
<code>-Z 'none'</code>	No support packages are included. Using this option can cause runtime errors.
<code>-Z packagename</code>	Only the specified support package is included. To specify multiple support packages, use multiple <code>-Z</code> inputs.

---

**Note** To list installed support packages or those used by a specific file, see `compiler.codetools.deployableSupportPackages`.

---

## Tips

- On Windows, you can generate a system-level file version number for your target file by appending `version=version_number` to the target generating `mcc` syntax. For an example, see “Create a C++ shared library with a system-level file version number (Windows only)”.

*version\_number* — Specifies the version of the target file as *major.minor.bug.build* in the file system. You are not required to specify a version number. If you do not specify a version number, `mcc` sets the version number, by default, to `1.0.0.0`.

- *major* — Specifies the major version number. If you do not specify a version number, `mcc` sets *major* to `1`.
- *minor* — Specifies the minor version number. If you do not specify a version number, `mcc` sets *minor* to `0`.
- *bug* — Specifies the bug fix maintenance release number. If you do not specify a version number, `mcc` sets *bug* to `0`.
- *build* — Specifies build number. If you do not specify a version number, `mcc` sets *build* to `0`.

This functionality is supported for C shared libraries, C++ shared libraries, COM components, .NET assemblies, and Excel add-ins for MATLAB Production Server in MATLAB Compiler SDK. For supported targets in MATLAB Compiler, see the **Tips** section in `mcc`.

## Version History

Introduced before R2006a

## See Also

`mbuild`

## mclGetLastErrorMessage

Last error message from unsuccessful library initialization or MATLAB function call

### Syntax

```
const char* mclGetLastErrorMessage()
```

### Description

This function returns a message corresponding to the most recent error encountered during library initialization or MATLAB function execution. If no error has occurred, the function returns an empty character array.

### Example

```
char *args[] = { "-nodisplay" };
if (!mclInitializeApplication(args, 1))
{
    /* Code here cannot use mclGetLastErrorMessage(), which only
       reports errors after mclInitializeApplication() succeeds. */
    fprintf(stderr, "An error occurred while initializing the application.")
    return -1;
}
if (!libmatrixInitialize())
{
    char * message = "An error occurred while initializing libmatrix";
    char * details = mclGetLastErrorMessage();
    if (details && *details)
    {
        fprintf(stderr, "%s: %s", message, details);
    }
    else
    {
        fprintf(stderr, message);
    }
    return -2;
}
```

## Version History

**Introduced in R2010b**

### See Also

[mclInitializeApplication](#) | [mclTerminateApplication](#) |  
[<library>Initialize\[WithHandlers\]](#) | [<library>Terminate](#)

# mclGetLogFileName

Retrieve name of log file used by MATLAB Runtime

## Syntax

```
const char* mclGetLogFileName()
```

## Description

Use `mclGetLogFileName()` to retrieve the name of the log file used by the MATLAB Runtime. The function returns a character string representing log file name used by MATLAB Runtime.

## Examples

```
printf("Logfile name : %s\n",mclGetLogFileName());
```

## Version History

Introduced in R2009a

## mclInitializeApplication

Set up application state shared by all MATLAB Runtime instances created in current process

### Syntax

```
bool mclInitializeApplication(const char **options, int count)
```

### Description

`bool mclInitializeApplication(const char **options, int count)` sets up the application state shared by all MATLAB Runtime instances created in the current process. The function takes an array of strings (possibly of zero length) specifying additional MATLAB Runtime options and a count specifying the size of the string array.

### Examples

#### Initialize MATLAB Runtime Instances

In the main function of your C/C++ application code, call `mclInitializeApplication` to start all MATLAB Runtime instances:

```
/* Call the mclInitializeApplication routine. Make sure that the application
 * was initialized properly by checking the return status. This initialization
 * has to be done before calling any MATLAB APIs or MATLAB Compiler SDK
 * generated shared library functions.
 */
if (!mclInitializeApplication(nullptr, 0))
{
    std::cerr << "Could not initialize the application." << std::endl;
    return -1;
}
```

---

**Caution** `mclInitializeApplication` must be called once only per process. Calling `mclInitializeApplication` more than once may cause your application to exhibit unpredictable or undesirable behavior.

---

#### Initialize MATLAB Runtime Instances Using Options

In the main function of your C/C++ application code, call `mclInitializeApplication` to start all MATLAB Runtime instances with the `-nodisplay` option:

```
const char *args[] = { "-nodisplay" };
if (! mclInitializeApplication(args, 1))
{
    fprintf(stderr,
            "An error occurred while initializing: \n %s ",
```

```
        mclGetLastErrorMessage());  
    return -1;  
}
```

## Input Arguments

### options — MATLAB Runtime options

NULL | string array

MATLAB Runtime options, specified as a string array. The string array may contain the following MATLAB command line switches, which have the same meaning as they do when used in MATLAB:

- -appendlogfile
- -Automation
- -beginfile
- -debug
- -defer
- -display
- -Embedding
- -endfile
- -fork
- -java
- -jdb
- -logfile
- -minimize
- -MLAutomation
- -nodisplay
- -noFigureWindows
- -nojvm
- -noselldde
- -nosplash
- -r
- -Regserver
- -selldde
- -singleCompThread
- -Unregserver
- -useJavaFigures
- -mwvisual
- -xrm

---

**Caution** When running on Mac, if -nodisplay is used as one of the options included in options, then the call to mclInitializeApplication must occur before calling mclRunMain.

---

Example: {"-singleCompThread", "-nodisplay"}

**count – Size of options string array**

NULL | string array

Size of the options string array, specified as an integer.

Example: 2

**Output Arguments****bool – Initialization result**

Initialization result, returned as a boolean value. Result indicates whether or not `mcli` initialization was successful. If the function returns `false`, calling any further compiled functions results in unpredictable behavior.

**Version History**

Introduced in R2009a

**See Also**

`mclTerminateApplication` | `<library>Initialize[WithHandlers]`

**Topics**

"Library Initialization and Termination Functions" on page 2-18

"Call a C Shared Library" on page 2-2



## mclIsJVMEnabled

Determine if MATLAB Runtime was started with instance of Java Virtual Machine (JVM)

### Syntax

```
bool mclIsJVMEnabled()
```

### Description

Use `mclIsJVMEnabled()` to determine if MATLAB Runtime was started with an instance of a Java Virtual Machine (JVM™). Returns `true` if MATLAB Runtime is started with a JVM instance, else returns `false`.

### Examples

```
printf("JVM initialized : %d\n", mclIsJVMEnabled());
```

### Version History

Introduced in R2009a

## mclIsMCRInitialized

Determine if MATLAB Runtime has been properly initialized

### Syntax

```
bool mclIsMCRInitialized()
```

### Description

Use `mclIsMCRInitialized()` to determine whether or not MATLAB Runtime has been properly initialized. Returns

- `true` if MATLAB Runtime is already initialized
- `false` if MATLAB Runtime is not initialized

---

**Note** This method can only be called once the MATLAB Runtime proxy library has been initiated.

---

### Examples

```
printf("MCR initialized : %d\n", mclIsMCRInitialized());
```

## Version History

**Introduced in R2009a**

## mclIsNoDisplaySet

Determine if -nodisplay mode is enabled

### Syntax

```
bool mclIsNoDisplaySet()
```

### Description

Use `mclIsNoDisplaySet()` to determine if -nodisplay mode is enabled. Returns `true` if -nodisplay is enabled, else returns `false`.

---

**Note** Always returns `false` on Windows systems, since the -nodisplay option is not supported on Windows systems.

---

### Examples

```
printf("nodisplay set : %d\n",mclIsNoDisplaySet());
```

## Version History

Introduced in R2009a

## **mclmcrInitialize**

Initialize the MATLAB Runtime proxy library

### **Syntax**

```
mclmcrInitialize()
```

### **Description**

`mclmcrInitialize()` initializes the library used to create the MATLAB Runtime proxy that is used by all other APIs generated by MATLAB Compiler SDK.

`mclmcrInitialize` is called by `mclInitializeApplication`. Therefore, in most cases, you should not explicitly call this function in your application code.

### **Version History**

**Introduced in R2013b**

### **See Also**

`mclInitializeApplication`

# mclRunMain

Mechanism for creating identical wrapper code across all platforms

## Syntax

```
typedef int (*mclMainFcnType)(int, const char **);  
  
int mclRunMain(mclMainFcnType run_main,  
              int argc,  
              const char **argv)
```

## Description

As you need to provide wrapper code when creating an application which uses a C or C++ shared library created by MATLAB Compiler SDK, `mclRunMain` enables you with a mechanism for creating identical wrapper code across all MATLAB Compiler SDK platform environments.

`mclRunMain` is especially helpful in Macintosh OS X environments where a run loop must be created for correct MATLAB Runtime operation.

When a Mac OS X run loop is started, if `mclInitializeApplication` specifies the `-nojvm` or `-nodisplay` option, creating a run loop is a straightforward process. Otherwise, you must create a Cocoa framework. The Cocoa frameworks consist of libraries, APIs, and MATLAB Runtime that form the development layer for all of Mac OS X.

Generally, the function pointed to by `run_main` returns with a pointer (return value) to the code that invoked it. However, when using Cocoa on the Macintosh, when the function pointed to by `run_main` returns, MATLAB Runtime calls `exit` before the return value can be received by the application because the underlying code cannot get control when Cocoa is shut down.

---

**Caution** You should not use `mclRunMain` if your application brings up its own full graphical environment.

---

---

**Note** In non-Macintosh environments, `mclRunMain` acts as a wrapper and does not perform any significant processing.

---

## Parameters

### `run_main`

Name of function to execute after MATLAB Runtime set-up code.

### `argc`

Number of arguments being passed to `run_main` function. Usually, `argc` is received by application at its `main` function.

**argv**

Pointer to an array of character pointers. Usually, argv is received by application at its main function.

**Examples**

Call using this basic structure:

```
int returncode = 0;
mclInitializeApplication(NULL,0);
returncode = mclRunMain((mclmainFcn)
    my_main_function,0,NULL);
```

**Version History**

Introduced in R2010b

**See Also**

mclInitializeApplication

# mclTerminateApplication

Close MATLAB Runtime-internal application state

## Syntax

```
bool mclTerminateApplication(void)
```

## Description

Call this function once at the end of your program to close MATLAB Runtime-internal application state. Call only once per process. After you have called this function, you cannot call any further MATLAB Compiler SDK generated functions or any functions in any MATLAB library.

---

**Caution** `mclTerminateApplication` must be called once only per process. Calling `mclTerminateApplication` more than once may cause your application to exhibit unpredictable or undesirable behavior.

---

---

**Caution** `mclTerminateApplication` closes any visible or invisible figures before exiting. If you have visible figures that you would like to wait for, use `mclWaitForFiguresToDie`.

---

## Examples

At the start of your program, call `mclInitializeApplication` to ensure that your library was properly initialized:

```
mclInitializeApplication(NULL,0);
if (!libmatrixInitialize()){
    fprintf(stderr,
            "An error occurred while initializing: \n %s ",
            mclGetLastErrorMessage());
    return -1;
}
```

At your program's exit point, call `mclTerminateApplication` to properly shut down the application:

```
mxDestroyArray(in1); in1=0;
mxDestroyArray(in2); in2 = 0;
mclTerminateApplication();
return 0;
```

## Version History

Introduced in R2009a

## See Also

`mclInitializeApplication`

## mclWaitForFiguresToDie

Enable deployed applications to process graphics events so that figure windows remain displayed

### Syntax

```
void mclWaitForFiguresToDie(HMCRINSTANCE instReserved)
```

### Description

Calling `void mclWaitForFiguresToDie` enables the deployed application to process graphics events.

NULL is the only parameter accepted for the MATLAB Runtime instance (HMCRINSTANCE `instReserved`).

This function can only be called after `libraryInitialize` has been called and before `libraryTerminate` has been called.

`mclWaitForFiguresToDie` blocks all open figures. This function runs until no visible figures remain. At that point, it displays a warning if there are invisible figures present. This function returns only when the last figure window is manually closed — therefore, this function should be called after the library runs at least one figure window. This function may be called multiple times.

If this function is not called, any figure windows initially displayed by the application briefly appear, and then the application exits.

---

**Note** `mclWaitForFiguresToDie` blocks the calling program only for MATLAB figures. It does not block any Java GUIs, ActiveX® controls, or other non-MATLAB GUIs unless they are embedded in a MATLAB figure window.

---

### Examples

```
int run_main(int argc, const char** argv)
{
    int some_variable = 0;
    if (argc > 1)
        test_to_run = atoi(argv[1]);

    /* Initialize application */

    if( !mclInitializeApplication(NULL,0) )
    {
        fprintf(stderr,
                "An error occurred while
                initializing: \n %s ",
                mclGetLastErrorMessage());
        return -1;
    }

    if (test_to_run == 1 || test_to_run == 0)
```



```

{
    /* Initialize axlks library */
    if (!libaxlksInitialize())
    {
        fprintf(stderr,
            "An error occurred while
            initializing: \n %s ",
            mclGetLastErrorMessage());
        return -1;
    }
}

if (test_to_run == 2 || test_to_run == 0)
{
    /* Initialize simple library */
    if (!libsimpleInitialize())
    {
        fprintf(stderr,
            "An error occurred while
            initializing: \n %s ",
            mclGetLastErrorMessage());
        return -1;
    }
}

/* your code here
/* your code here
/* your code here
/* your code here
/*
/* Block on open figures */
mclWaitForFiguresToDie(NULL);
/* Terminate libraries */
if (test_to_run == 1 || test_to_run == 0)
    libaxlksTerminate();
if (test_to_run == 2 || test_to_run == 0)
    libsimplifyTerminate();
/* Terminate application */
mclTerminateApplication();
return(0);
}

```

## Version History

Introduced in R2009a

### See Also

[mclInitializeApplication](#) | [mclRunMain](#) | [mclTerminateApplication](#)

## <library>Terminate

Free all resources allocated by MATLAB Runtime instance associated with *library*

### Syntax

```
libraryTerminate()
```

### Description

`libraryTerminate()` frees all resources allocated by the MATLAB Runtime instance associated with the generated C/C++ shared library *library*. Call this function in your C/C++ application after you finish calling the functions in this generated library and before calling `mclTerminateApplication`.

### Examples

#### Terminate Library

Terminate a C shared library named `libmatrix`.

- 1 Call the initialization function for the C library `libmatrix` in the main function of your C application code.

```

...
/* Call the library initialization routine and ensure the
 * library was initialized properly. */
if (!libmatrixInitialize())
{
    fprintf(stderr,
            "An error occurred while initializing: \n %s ",
            mclGetLastErrorMessage());
    return -2;
}
else
{
    /* Call the library function(s) for your application */
    ...

```

- 2 At the end of the else statement (before calling `mclTerminateApplication`), call `libmatrixTerminate` to free resources allocated by the MATLAB Runtime instance associated with `libmatrix`.

```

...
/* Call the library termination routine */
libmatrixTerminate();
/* Free the memory created */
mxDestroyArray(in1); in1=0;
mxDestroyArray(in2); in2 = 0;
}

```

- 3 Finally, call `mclTerminateApplication` to terminate the MATLAB Runtime instance associated with `libmatrix`.

```
/* mclTerminateApplication shuts down MATLAB Runtime.
 * You cannot restart it by calling mclInitializeApplication.
 * Call mclTerminateApplication once and only once in your application.
 */
mclTerminateApplication();
return 0;
}
```

## Input Arguments

### Library — Library name

Library name, specified as part of the function name. The library must be a C/C++ shared library generated by MATLAB Compiler SDK.

Example: `libmatrix`

## Version History

Introduced in R2015a

### See Also

<library>Initialize[WithHandlers] | mclTerminateApplication

### Topics

“Library Initialization and Termination Functions” on page 2-18

“Call a C Shared Library” on page 2-2



# C++ Utility Library Reference

## **Data Conversion Restrictions for the C++ mxArray API**

Currently, returning a Java object to your application, from a compiled MATLAB function, is unsupported.

## Primitive Types

The `mwArray` API supports all primitive types that can be stored in a MATLAB array. This table lists all the types.

Type	Description	mxClassID
<code>mxChar</code>	Character type	<code>mxCHAR_CLASS</code>
<code>mxLogical</code>	Logical or Boolean type	<code>mxLOGICAL_CLASS</code>
<code>mxDouble</code>	Double-precision floating-point type	<code>mxDOUBLE_CLASS</code>
<code>mxSingle</code>	Single-precision floating-point type	<code>mxSINGLE_CLASS</code>
<code>mxInt8</code>	1-byte signed integer	<code>mxINT8_CLASS</code>
<code>mxUInt8</code>	1-byte unsigned integer	<code>mxUINT8_CLASS</code>
<code>mxInt16</code>	2-byte signed integer	<code>mxINT16_CLASS</code>
<code>mxUInt16</code>	2-byte unsigned integer	<code>mxUINT16_CLASS</code>
<code>mxInt32</code>	4-byte signed integer	<code>mxINT32_CLASS</code>
<code>mxUInt32</code>	4-byte unsigned integer	<code>mxUINT32_CLASS</code>
<code>mxInt64</code>	8-byte signed integer	<code>mxINT64_CLASS</code>
<code>mxUInt64</code>	8-byte unsigned integer	<code>mxUINT64_CLASS</code>

## **C++ Utility Classes**

- mwString
- mwException
- mwArray



# mwString

String class used by the mwArray API to pass string data as output from certain methods

## Description

The mwString class is a simple string class used by the mwArray API to pass string data as output from certain methods.

## Required Headers

- mclcppclass.h
- mclmcrnt.h

---

**Tip** MATLAB Compiler SDK automatically includes these header files in the header file generated for your MATLAB functions.

---

## Constructors

### mwString()

#### Description

Create an empty string.

### mwString(char\* str)

#### Description

Create a new string and initialize the string's data with the supplied char buffer.

#### Arguments

char* str	Null terminated character buffer
-----------	----------------------------------

### mwString(mwString& str)

#### Description

Create a new string and initialize the string's data with the contents of the supplied string.

#### Arguments

mwString& str	Initialized mwString instance
---------------	-------------------------------

## Methods

### int Length() const

#### Description

Return the number of characters in string.

**Example**

```
mwString str("This is a string");  
int len = str.Length();
```

**Operators****operator const char\* () const****Description**

Return a pointer to internal buffer of string.

**Example**

```
mwString str("This is a string");  
const char* pstr = (const char*)str;
```

**mwString& operator=(const mwString& str)****Description**

Copy the contents of one string into a new string.

**Arguments**

mwString& str	Initialized mwString instance to copy
---------------	---------------------------------------

**Example**

```
mwString str("This is a string");  
mwString new_str = str;
```

**mwString& operator=(const char\* str)****Description**

Copy the contents of a null terminated character buffer into a new string.

**Arguments**

char* str	Null terminated character buffer to copy
-----------	--

**Example**

```
const char* pstr = "This is a string";  
mwString str = pstr;
```

**bool operator==(const mwString& str) const****Description**

Test two mwString instances for equality. If the characters in the string are the same, the instances are equal.

**Arguments**

mwString& str	Initialized mwString instance
---------------	-------------------------------

**Example**

```
mwString str("This is a string");
mwString str2("This is another string");
bool ret = (str == str2);
```

**bool operator!=(const mwString& str) const****Description**

Test two `mwString` instances for inequality. If the characters in the string are not the same, the instances are unequal.

**Arguments**

<code>mwString&amp; str</code>	Initialized <code>mwString</code> instance
--------------------------------	--

**Example**

```
mwString str("This is a string");
mwString str2("This is another string");
bool ret = (str != str2);
```

**bool operator<(const mwString& str) const****Description**

Compare two strings and return `true` if the first string is lexicographically less than the second string.

**Arguments**

<code>mwString&amp; str</code>	Initialized <code>mwString</code> instance
--------------------------------	--

**Example**

```
mwString str("This is a string");
mwString str2("This is another string");
bool ret = (str < str2);
```

**bool operator<=(const mwString& str) const****Description**

Compare two strings and return `true` if the first string is lexicographically less than or equal to the second string.

**Arguments**

<code>mwString&amp; str</code>	Initialized <code>mwString</code> instance
--------------------------------	--

**Example**

```
mwString str("This is a string");
mwString str2("This is another string");
bool ret = (str <= str2);
```

**bool operator>(const mwString& str) const****Description**

Compare two strings and return `true` if the first string is lexicographically greater than the second string.

**Arguments**

<code>mwString&amp; str</code>	Initialized <code>mwString</code> instance
--------------------------------	--

**Example**

```
mwString str("This is a string");
mwString str2("This is another string");
bool ret = (str > str2);
```

**bool operator>=(const mwString& str) const****Description**

Compare two strings and return `true` if the first string is lexicographically greater than or equal to the second string.

**Arguments**

<code>mwString&amp; str</code>	Initialized <code>mwString</code> instance
--------------------------------	--

**Example**

```
mwString str("This is a string");
mwString str2("This is another string");
bool ret = (str >= str2);
```

**friend std::ostream& operator<<(std::ostream& os, const mwString& str)****Description**

Copy contents of input string to specified `ostream`.

**Arguments**

<code>std::ostream&amp; os</code>	Initialized <code>ostream</code> instance to copy string into
<code>mwString&amp; str</code>	Initialized <code>mwString</code> instance to copy

**Example**

```
#include <ostream>
mwString str("This is a string");
std::cout << str << std::endl;
```

**Version History**

Introduced in R2013b

# mwException

Exception type used by the `mwArray` API and the C++ interface functions

## Description

The `mwException` class is the basic exception type used by the `mwArray` API and the C++ interface functions. All errors created during calls to the `mwArray` API and to generated C++ interface functions are thrown as `mwExceptions`.

## Required Headers

- `mclcppclass.h`
- `mclmcrst.h`

---

**Tip** MATLAB Compiler SDK automatically includes these header files in the header file generated for your MATLAB functions.

---

## Constructors

### `mwException()`

#### Description

Construct new `mwException` with default error message.

### `mwException(char* msg)`

#### Description

Create an `mwException` with a specified error message.

#### Arguments

<code>char* msg</code>	Null terminated character buffer to use as the error message
------------------------	--

### `mwException(mwException& e)`

#### Description

Create a copy of an `mwException`.

#### Arguments

<code>mwException&amp; e</code>	Initialized <code>mwException</code> instance to copy
---------------------------------	---

### `mwException(std::exception& e)`

#### Description

Create new `mwException` from existing `std::exception`.

**Arguments**

std::exception& e	std::exception to copy
-------------------	------------------------

**Methods****char\* what() const throw()****Description**

Return the error message contained in this exception.

**Example**

```
try
{
    ...
}
catch (const std::exception& e)
{
    std::cout << e.what() << std::endl;
}
```

**void print\_stack\_trace()****Description**

Print the stack trace to std::cerr.

**Operators****mwException& operator=(const mwException& e)****Description**

Copy the contents of one exception into a new exception.

**Arguments**

mwException& e	An initialized mwException instance to copy
----------------	---

**Example**

```
try
{
    ...
}
catch (const mwException& e)
{
    mwException e2 = e;
    throw e2;
}
```

**mwException& operator=(const std::exception& e)****Description**

Copy the contents of one exception into a new exception.

**Arguments**

<code>std::exception&amp; e</code>	<code>std::exception</code> to copy
------------------------------------	-------------------------------------

**Example**

```
try
{
    ...
}
catch (const std::exception& e)
{
    mwException e2 = e;
    throw e2;
}
```

**Version History****Introduced in R2013b**

# mwArray

Class used to pass input/output arguments to C++ functions generated by MATLAB Compiler SDK

## Description

Use the `mwArray` class to pass input/output arguments to generated C++ interface functions. This class consists of a thin wrapper around a MATLAB array. All data in MATLAB is represented by arrays. The `mwArray` class provides the necessary constructors, methods, and operators for array creation and initialization, as well as simple indexing.

---

**Note** Arithmetic operators, such as addition and subtraction, are no longer supported as of Release 14.

---

## Required Headers

- `mclcppclass.h`
- `mclmcrct.h`

---

**Tip** MATLAB Compiler SDK automatically includes these header files in the header file generated for your MATLAB functions.

---

## Constructors

### `mwArray()`

#### Description

Construct empty array of type `mxDOUBLE_CLASS`.

### `mwArray(mxClassID mxID)`

#### Description

Construct empty array of specified type.

#### Arguments

<code>mxClassID mxID</code>	Valid <code>mxClassID</code> specifying the type of array to construct. See “Work with mxArray” for more information on <code>mxClassID</code> .
-----------------------------	--

### `mwArray(mwSize num_rows, mwSize num_cols, mxClassID mxID, mxComplexity cmplx = mxREAL)`

#### Description

Create a 2-D matrix of the specified type and complexity. For nonnumeric types, `mxComplexity` will be ignored. For numeric types, pass `mxCOMPLEX` for the last argument to create a complex matrix;



otherwise, the matrix will be real. All elements are initialized to zero. For cell matrices, all elements are initialized to empty cells.

#### Arguments

<code>mwSize num_rows</code>	Number of rows in the array
<code>mwSize num_cols</code>	Number of columns in the array
<code>mxClassID mxID</code>	Valid <code>mxClassID</code> specifying the type of array to construct. See “Work with mxArrays” for more information on <code>mxClassID</code> .
<code>mxComplexity cmplx</code>	Complexity of the array to create. Valid values are <code>mxREAL</code> and <code>mxCOMPLEX</code> . The default value is <code>mxREAL</code> .

#### **`mwArray(mwSize num_dims, const mwSize* dims, mxClassID mxID, mxComplexity cmplx = mxREAL)`**

##### Description

Create an n-dimensional array of the specified type and complexity. For nonnumeric types, `mxComplexity` will be ignored. For numeric types, pass `mxCOMPLEX` for the last argument to create a complex matrix; otherwise, the array will be real. All elements are initialized to zero. For cell arrays, all elements are initialized to empty cells.

#### Arguments

<code>mwSize num_dims</code>	Number of dimensions in the array
<code>const mwSize* dims</code>	Dimensions of the array
<code>mxClassID mxID</code>	Valid <code>mxClassID</code> specifying the type of array to construct. See “Work with mxArrays” for more information on <code>mxClassID</code> .
<code>mxComplexity cmplx</code>	Complexity of the array to create. Valid values are <code>mxREAL</code> and <code>mxCOMPLEX</code> . The default value is <code>mxREAL</code> .

#### **`mwArray(const char* str)`**

##### Description

Create a 1-by-*n* array of type `mxCHAR_CLASS`, with  $n = \text{strlen}(\text{str})$ , and initialize the array's data with the characters in the supplied string.

#### Arguments

<code>const char* str</code>	Null-terminated character buffer used to initialize the array
------------------------------	---

#### **`mwArray(mwSize num_strings, const char** str)`**

##### Description

Create a matrix of type `mxCHAR_CLASS`, and initialize the array's data with the characters in the supplied strings. The created array has dimensions *m*-by-*max*, where *m* is the number of strings and *max* is the length of the longest string in `str`.

**Arguments**

mwSize num_strings	Number of strings in the input array
const char** str	Array of null-terminated strings

**mwArray(mwSize num\_rows, mwSize num\_cols, int num\_fields, const char\*\* fieldnames)****Description**

Create a matrix of type mxSTRUCT\_CLASS, with the specified field names. All elements are initialized with empty cells.

**Arguments**

mwSize num_rows	Number of rows in the array
mwSize num_cols	Number of columns in the array
int num_fields	Number of fields in the struct matrix.
const char** fieldnames	Array of null-terminated strings representing the field names

**mwArray(mwSize num\_dims, const mwSize\* dims, int num\_fields, const char\*\* fieldnames)****Description**

Create an n-dimensional array of type mxSTRUCT\_CLASS, with the specified field names. All elements are initialized with empty cells.

**Arguments**

mwSize num_dims	Number of dimensions in the array
const mwSize* dims	Dimensions of the array
int num_fields	Number of fields in the struct matrix.
const char** fieldnames	Array of null-terminated strings representing the field names

**mwArray(const mxArray& arr)****Description**

Create a deep copy of an existing array.

**Arguments**

mwArray& arr	mwArray to copy
--------------	-----------------

**mwArray(<type> re)****Description**

Create a real scalar array.

The scalar array is created with the type of the input argument.

**Arguments**

<type> re	Scalar value to initialize the array. <type> can be any of the following: <ul style="list-style-type: none"> <li>• mxDouble</li> <li>• mxSingle</li> <li>• mxInt8</li> <li>• mxUInt8</li> <li>• mxInt16</li> <li>• mxUInt16</li> <li>• mxInt32</li> <li>• mxUInt32</li> <li>• mxInt64</li> <li>• mxUInt64</li> <li>• mxLogical</li> </ul>
-----------	---

**mwArray(<type> re, <type> im)****Description**

Create a complex scalar array.

The scalar array is created with the type of the input argument.

**Arguments**

<type> re	Scalar value to initialize the real part of the array
<type> im	Scalar value to initialize the imaginary part of the array

<type> can be any of the following:

- mxDouble
- mxSingle
- mxInt8
- mxUInt8
- mxInt16
- mxUInt16
- mxInt32
- mxUInt32
- mxInt64
- mxUInt64
- mxLogical

## Methods

### **mwArray Clone() const**

#### **Description**

Create a new array representing deep copy of array.

#### **Example**

```
mwArray a(2, 2, mxDOUBLE_CLASS);  
mwArray b = a.Clone();
```

### **mwArray SharedCopy() const**

#### **Description**

Create a shared copy of an existing array. The new array and the original array both point to the same data.

#### **Example**

```
mwArray a(2, 2, mxDOUBLE_CLASS);  
mwArray b = a.SharedCopy();
```

### **mwArray Serialize() const**

#### **Description**

Serialize an array into bytes. A 1-by-n numeric matrix of type `mxUINT8_CLASS` is returned containing the serialized data. The data can be deserialized back into the original representation by calling `mwArray::Deserialize()`.

#### **Example**

```
mwArray a(2, 2, mxDOUBLE_CLASS);  
mwArray b = a.Serialize();
```

### **mxClassID ClassID() const**

#### **Description**

Determine the type of the array. See “Work with mxArray” for more information on `mxClassID`.

#### **Example**

```
mwArray a(2, 2, mxDOUBLE_CLASS);  
mxClassID id = a.ClassID();
```

### **size\_t ElementSize() const**

#### **Description**

Determine the size, in bytes, of an element of array type. If the array is complex, the return value will represent the size, in bytes, of the real part of an element.

#### **Example**

```
mwArray a(2, 2, mxDOUBLE_CLASS);  
int size = a.ElementSize();
```

**mwSize NumberOfElements() const****Description**

Determine the total size of the array.

**Example**

```
mwArray a(2, 2, mxDOUBLE_CLASS);  
int n = a.NumberOfElements();
```

**mwSize NumberOfNonZeros() const****Description**

Determine the size of the array's data. If the underlying array is not sparse, this returns the same value as `NumberOfElements()`.

**Example**

```
mwArray a(2, 2, mxDOUBLE_CLASS);  
int n = a.NumberOfNonZeros();
```

**mwSize MaximumNonZeros() const****Description**

Determine the allocated size of the array's data. If the underlying array is not sparse, this returns the same value as `NumberOfElements()`.

**Example**

```
mwArray a(2, 2, mxDOUBLE_CLASS);  
int n = a.MaximumNonZeros();
```

**mwSize NumberOfDimensions() const****Description**

Determine the dimensionality of the array.

**Example**

```
mwArray a(2, 2, mxDOUBLE_CLASS);  
int n = a.NumberOfDimensions();
```

**int NumberOfFields() const****Description**

Determine the number of fields in a `struct` array. If the underlying array is not of type `struct`, zero is returned.

**Example**

```
const char* fields[] = {"a", "b", "c"};  
mwArray a(2, 2, 3, fields);  
int n = a.NumberOfFields();
```

**mwString GetFieldName(int index)****Description**

Determine the name of a given field in a `struct` array. If the underlying array is not of type `struct`, an exception is thrown.

**Arguments**

<code>int index</code>	Index of the field to name. Indexing starts at zero.
------------------------	--

**Example**

```
const char* fields[] = {"a", "b", "c"};
mwArray a(2, 2, 3, fields);
mwString tempname = a.GetFieldName(1);
const char* name = (const char*)tempname;
```

**mwArray GetDimensions() const****Description**

Determine the size of each dimension in the array. The size of the returned array is 1-by-`NumberOfDimensions()`.

**Example**

```
mwArray a(2, 2, mxDOUBLE_CLASS);
mwArray dims = a.GetDimensions();
```

**bool IsEmpty() const****Description**

Determine if an array is empty.

**Example**

```
mwArray a;
bool b = a.IsEmpty();
```

**bool IsSparse() const****Description**

Determine if an array is sparse.

**Example**

```
mwArray a(2, 2, mxDOUBLE_CLASS);
bool b = a.IsSparse();
```

**bool IsNumeric() const****Description**

Determine if an array is numeric.

**Example**

```
mwArray a(2, 2, mxDOUBLE_CLASS);
bool b = a.IsNumeric();
```

**bool IsComplex() const****Description**

Determine if an array is complex.

**Example**

```
mwArray a(2, 2, mxDOUBLE_CLASS, mxCOMPLEX);
bool b = a.IsComplex();
```

**bool Equals(const mwArray& arr) const****Description**

Returns `true` if the input array is byte-wise equal to this array. This method makes a byte-wise comparison of the underlying arrays. Therefore, arrays of the same type should be compared. Arrays of different types will not in general be equal, even if they are initialized with the same data.

**Arguments**

mwArray& arr	Array to compare to array.
--------------	----------------------------

**Example**

```
mwArray a(1, 1, mxDOUBLE_CLASS);
mwArray b(1, 1, mxDOUBLE_CLASS);
a = 1.0;
b = 1.0;
bool c = a.Equals(b);
```

**int CompareTo(const mwArray& arr) const****Description**

Compares this array with the specified array for order. This method makes a byte-wise comparison of the underlying arrays. Therefore, arrays of the same type should be compared. Arrays of different types will, in general, not be ordered equivalently, even if they are initialized with the same data.

**Arguments**

mwArray& arr	Array to compare to array.
--------------	----------------------------

**Example**

```
mwArray a(1, 1, mxDOUBLE_CLASS);
mwArray b(1, 1, mxDOUBLE_CLASS);
a = 1.0;
b = 1.0;
int n = a.CompareTo(b);
```

**int GetHashCode() const****Description**

Constructs a unique hash value from the underlying bytes in the array. Therefore, arrays of different types will have different hash codes, even if they are initialized with the same data.

**Example**

```
mwArray a(1, 1, mxDOUBLE_CLASS);
int n = a.GetHashCode();
```

**mwString ToString() const****Description**

Returns a string representation of the underlying array. The string returned is the same one that is returned by typing a variable's name at the MATLAB command prompt.

**Example**

```
mwArray a(1, 1, mxDOUBLE_CLASS, mxCOMPLEX);
a.Real() = 1.0;
a.Imag() = 2.0;
printf("%s\n", (const char*)(a.ToString()));
```

**mwArray RowIndex() const****Description**

Returns an array representing the row indices (first dimension) of the elements of this array in column-major order. For sparse arrays, the indices are returned for just the non-zero elements and the size of the array returned is 1-by-NumberOfNonZeros(). For nonsparse arrays, the size of the array returned is 1-by-NumberOfElements(), and the row indices of all of the elements are returned.

**Example**

```
#include <stdio.h>
mwArray a(1, 1, mxDOUBLE_CLASS);
mwArray rows = a.RowIndex();
```

**mwArray ColumnIndex() const****Description**

Returns an array representing the column indices (second dimension) of the elements of this array in column-major order. For sparse arrays, the indices are returned for just the non-zero elements and the size of the array returned is 1-by-NumberOfNonZeros(). For nonsparse arrays, the size of the array returned is 1-by-NumberOfElements(), and the column indices of all of the elements are returned.

**Example**

```
mwArray a(1, 1, mxDOUBLE_CLASS);
mwArray rows = a.ColumnIndex();
```

**void MakeComplex()****Description**

Convert a numeric array that has been previously allocated as `real` to `complex`. If the underlying array is of a nonnumeric type, an `mwException` is thrown.

**Example**

```
double rdata[4] = {1.0, 2.0, 3.0, 4.0};
double idata[4] = {10.0, 20.0, 30.0, 40.0};
```



```
mwArray a(2, 2, mxDOUBLE_CLASS);
a.SetData(rdata, 4);
a.MakeComplex();
a.Imag().SetData(idata, 4);
```

### **mwArray Get(mwSize num\_indices, ...)**

#### **Description**

Fetches a single element at a specified index. The number of indices is passed, followed by a comma-separated list of 1-based indices. The valid number of indices that can be passed in is either 1 (single subscript indexing) or `NumberOfDimensions()` (multiple subscript indexing). In single subscript indexing the element at the specified 1-based offset is returned, accessing data in column-major order. In multiple subscript indexing the index list is used to access the specified element. The valid range for indices is  $1 \leq \text{index} \leq \text{NumberOfElements}()$ , for single subscript indexing. For multiple subscript indexing, the  $i$ th index has the valid range:  $1 \leq \text{index}[i] \leq \text{GetDimensions}().\text{Get}(1, i)$ . An `mwException` is thrown if an invalid number of indices is passed in or if any index is out of bounds.

#### **Arguments**

<code>mwSize num_indices</code>	Number of indices passed in
<code>...</code>	Comma-separated list of input indices. Number of items must equal <code>num_indices</code> but should not exceed 32.

#### **Example**

```
double data[4] = {1.0, 2.0, 3.0, 4.0};
double x;
mwArray a(2, 2, mxDOUBLE_CLASS);
a.SetData(data, 4);
x = a.Get(1,1);
x = a.Get(2, 1, 2);
x = a.Get(2, 2, 2);
```

### **mwArray Get(const char\* name, mwSize num\_indices, ...)**

#### **Description**

Fetches a single element at a specified field name and index. This method may only be called on an array that is of type `mxSTRUCT_CLASS`. An `mwException` is thrown if the underlying array is not a `struct` array. The field name passed must be a valid field name in the `struct` array. The index is passed by first passing the number of indices followed by a comma-separated list of 1-based indices. The valid number of indices that can be passed in is either 1 (single subscript indexing) or `NumberOfDimensions()` (multiple subscript indexing). In single subscript indexing the element at the specified 1-based offset is returned, accessing data in column-wise order. In multiple subscript indexing the index list is used to access the specified element. The valid range for indices is  $1 \leq \text{index} \leq \text{NumberOfElements}()$ , for single subscript indexing. For multiple subscript indexing, the  $i$ th index has the valid range:  $1 \leq \text{index}[i] \leq \text{GetDimensions}().\text{Get}(1, i)$ . An `mwException` is thrown if an invalid number of indices is passed in or if any index is out of bounds.

**Arguments**

char* name	Null-terminated character buffer containing the name of the field
mwSize num_indices	Number of indices passed in
...	Comma-separated list of input indices. Number of items must equal num_indices but should not exceed 32.

**Example**

```
const char* fields[] = {"a", "b", "c"};

mwArray a(1, 1, 3, fields);
mwArray b = a.Get("a", 1, 1);
mwArray b = a.Get("b", 2, 1, 1);
```

**mwArray Real()****Description**

Accesses the real part of a complex array. The returned mxArray is considered real and has the same dimensionality and type as the original.

Complex arrays consist of Complex numbers, which are 1-by-2 vectors (pairs). For example, if the number is  $3+5i$ , then the pair is  $(3, 5i)$ . An array of Complex numbers is therefore two dimensional (N-by-2), where N is the number of complex numbers in the array.  $2+4i$ ,  $7-3i$ ,  $8+6i$  would be represented as  $(2, 4i)$   $(7, 3i)$   $(8, 6i)$ . Complex numbers have two components, real and imaginary.

**Example**

```
double rdata[4] = {1.0, 2.0, 3.0, 4.0};
double idata[4] = {10.0, 20.0, 30.0, 40.0};
mwArray a(2, 2, mxDOUBLE_CLASS, mxCOMPLEX);
a.Real().SetData(rdata, 4);
```

**mwArray Imag()****Description**

Accesses the imaginary part of a complex array. The returned mxArray is considered real and has the same dimensionality and type as the original.

Complex arrays consist of Complex numbers, which are 1-by-2 vectors (pairs). For example, if the number is  $3+5i$ , then the pair is  $(3, 5i)$ . An array of Complex numbers is therefore two dimensional (N-by-2), where N is the number of complex numbers in the array.  $2+4i$ ,  $7-3i$ ,  $8+6i$  would be represented as  $(2, 4i)$   $(7, 3i)$   $(8, 6i)$ . Complex numbers have two components, real and imaginary.

**Example**

```
double rdata[4] = {1.0, 2.0, 3.0, 4.0};
double idata[4] = {10.0, 20.0, 30.0, 40.0};
mwArray a(2, 2, mxDOUBLE_CLASS, mxCOMPLEX);
a.Imag().SetData(idata, 4);
```

**void Set(const mwArray& arr)****Description**

Assign shared copy of input array to currently referenced cell for arrays of type mxCELL\_CLASS and mxSTRUCT\_CLASS.

**Arguments**

mwArray& arr	mwArray to assign to currently referenced cell
--------------	--

**Example**

```
mwArray a(2, 2, mxDOUBLE_CLASS);
mwArray b(2, 2, mxINT16_CLASS);
mwArray c(1, 2, mxCELL_CLASS);
c.Get(1,1).Set(a);
c.Get(1,2).Set(b);
```

**void GetData(<numeric-type>\* buffer, mwSize len) const****Description**

Copies the array's data into supplied numeric buffer.

The data is copied in column-major order. If the underlying array is not of the same type as the input buffer, the data is converted to this type as it is copied. If a conversion cannot be made, an mxException is thrown.

**Arguments**

<numeric-type>* buffer	Buffer to receive copy. Valid types for <numeric-type> are: <ul style="list-style-type: none"> <li>• mxDOUBLE_CLASS</li> <li>• mxSINGLE_CLASS</li> <li>• mxINT8_CLASS</li> <li>• mxUINT8_CLASS</li> <li>• mxINT16_CLASS</li> <li>• mxUINT16_CLASS</li> <li>• mxINT32_CLASS</li> <li>• mxUINT32_CLASS</li> <li>• mxINT64_CLASS</li> <li>• mxUINT64_CLASS</li> </ul>
mwSize len	Maximum length of buffer. A maximum of len elements will be copied.

**Example**

```
double rdata[4] = {1.0, 2.0, 3.0, 4.0};
double data_copy[4];
mwArray a(2, 2, mxDOUBLE_CLASS);
a.SetData(rdata, 4);
a.GetData(data_copy, 4);
```

**void GetLogicalData(mxLogical\* buffer, mwSize len) const****Description**

Copies the array's data into supplied `mxLogical` buffer.

The data is copied in column-major order. If the underlying array is not of type `mxLOGICAL_CLASS`, the data is converted to this type as it is copied. If a conversion cannot be made, an `mwException` is thrown.

**Arguments**

<code>mxLogical* buffer</code>	Buffer to receive copy
<code>mwSize len</code>	Maximum length of buffer. A maximum of <code>len</code> elements will be copied.

**Example**

```
mxLogical data[4] = {true, false, true, false};
mxLogical data_copy[4] ;
mwArray a(2, 2, mxLOGICAL_CLASS);
a.SetLogicalData(data, 4);
a.GetLogicalData(data_copy, 4);
```

**void GetCharData(mxChar\* buffer, mwSize len) const****Description**

Copies the array's data into supplied `mxChar` buffer.

The data is copied in column-major order. If the underlying array is not of type `mxCHAR_CLASS`, the data is converted to this type as it is copied. If a conversion cannot be made, an `mwException` is thrown.

**Arguments**

<code>mxChar** buffer</code>	Buffer to receive copy
<code>mwSize len</code>	Maximum length of buffer. A maximum of <code>len</code> elements will be copied.

**Example**

```
mxChar data[6] = {'H', 'e', '\l', 'l', 'o', '\0'};
mxChar data_copy[6] ;
mwArray a(1, 6, mxCHAR_CLASS);
a.SetCharData(data, 6);
a.GetCharData(data_copy, 6);
```

**void SetData(<numeric-type>\* buffer, mwSize len)****Description**

Copies the data from supplied numeric buffer into the array.

The data is copied in column-major order. If the underlying array is not of the same type as the input buffer, the data is converted to this type as it is copied. If a conversion cannot be made, an `mwException` is thrown.

You cannot use `SetData` to dynamically resize an `mwArray`.

### Arguments

<numeric-type>* buffer	Buffer containing data to copy. Valid types for <numeric-type> are: <ul style="list-style-type: none"> <li>• mxDOUBLE_CLASS</li> <li>• mxSINGLE_CLASS</li> <li>• mxINT8_CLASS</li> <li>• mxUINT8_CLASS</li> <li>• mxINT16_CLASS</li> <li>• mxUINT16_CLASS</li> <li>• mxINT32_CLASS</li> <li>• mxUINT32_CLASS</li> <li>• mxINT64_CLASS</li> <li>• mxUINT64_CLASS</li> </ul>
mwSize len	Maximum length of buffer. A maximum of len elements will be copied.

### Example

```
double rdata[4] = {1.0, 2.0, 3.0, 4.0};
double data_copy[4] ;
mwArray a(2, 2, mxDOUBLE_CLASS);
a.SetData(rdata, 4);
a.GetData(data_copy, 4);
```

### void SetLogicalData(mxLogical\* buffer, mwSize len)

#### Description

Copies the data from the supplied `mxLogical` buffer into the array.

The data is copied in column-major order. If the underlying array is not of type `mxLOGICAL_CLASS`, the data is converted to this type as it is copied. If a conversion cannot be made, an `mwException` is thrown.

### Arguments

mxLogical* buffer	Buffer containing data to copy
mwSize len	Maximum length of buffer. A maximum of len elements will be copied.

### Example

```
mxLogical data[4] = {true, false, true, false};
mxLogical data_copy[4] ;
mwArray a(2, 2, mxLOGICAL_CLASS);
a.SetLogicalData(data, 4);
a.GetLogicalData(data_copy, 4);
```

**void SetCharData(mxChar\* buffer, mwSize len)****Description**

Copies the data from the supplied mxChar buffer into the array.

The data is copied in column-major order. If the underlying array is not of type mxCHAR\_CLASS, the data is converted to this type as it is copied. If a conversion cannot be made, an mxArrayException is thrown.

**Arguments**

mxChar** buffer	Buffer containing data to copy
mwSize len	Maximum length of buffer. A maximum of len elements will be copied.

**Example**

```
mxChar data[6] = {'H', 'e', '\l', 'l', 'o', '\0'};
mxChar data_copy[6];
mwArray a(1, 6, mxCHAR_CLASS);
a.SetCharData(data, 6);
a.GetCharData(data_copy, 6);
```

**static mxArray Deserialize(const mxArray& arr)****Description**

Deserializes an array that has been serialized with mxArray::Serialize(). The input array must be of type mxUINT8\_CLASS and contain the data from a serialized array. If the input data does not represent a serialized mxArray, the behavior of this method is undefined.

**Arguments**

mxArray& arr	mxArray that has been obtained by calling mxArray::Serialize
--------------	--

**Example**

```
double rdata[4] = {1.0, 2.0, 3.0, 4.0};
mwArray a(1,4,mxDOUBLE_CLASS);
a.SetData(rdata, 4);
mwArray b = a.Serialize();
a = mxArray::Deserialize(b);
```

**static mxArray NewSparse(mwSize rowindex\_size, const mwIndex\* rowindex, mwSize colindex\_size, const mwIndex\* colindex, mwSize data\_size, const mxDouble\* rdata, mwSize num\_rows, mwSize num\_cols, mwSize nzmax)****Description**

Creates real sparse matrix of type double with specified number of rows and columns.

The lengths of input row, column index, and data arrays must all be the same or equal to 1. In the case where any of these arrays are equal to 1, the value is repeated throughout the construction of the matrix.

If the same row/column pair occurs more than once, the data value assigned to that element is the sum of all values associated with that pair. If any element of the `rowindex` or `colindex` array is greater than the specified values in `num_rows` or `num_cols` respectively, an exception is thrown.

### Arguments

<code>mwSize rowindex_size</code>	Size of <code>rowindex</code> array
<code>mwIndex* rowindex</code>	Array of row indices of non-zero elements
<code>mwSize colindex_size</code>	Size of <code>colindex</code> array
<code>mwIndex* colindex</code>	Array of column indices of non-zero elements
<code>mwSize data_size</code>	Size of data array
<code>mxDouble* rdata</code>	Data associated with non-zero row and column indices
<code>mwSize num_rows</code>	Number of rows in matrix
<code>mwSize num_cols</code>	Number of columns in matrix
<code>mwSize nzmax</code>	Reserved storage for sparse matrix. If <code>nzmax</code> is zero, storage will be set to $\max\{\text{rowindex\_size}, \text{colindex\_size}, \text{data\_size}\}$ .

### Example

This example constructs a sparse 4-by-4 tridiagonal matrix:

```
2 -1 0 0
-1 2 -1 0
0 -1 2 -1
0 0 -1 2
```

The following code, when run:

```
double rdata[] =
    {2.0, -1.0, -1.0, 2.0, -1.0,
     -1.0, 2.0, -1.0, -1.0, 2.0};
mwIndex row_tridiag[] =
    {1, 2, 1, 2, 3,
     2, 3, 4, 3, 4 };
mwIndex col_tridiag[] =
    {1, 1, 2, 2, 2,
     3, 3, 3, 4, 4 };

mwArray mysparse =
    mwArray::NewSparse(10, row_tridiag,
                      10, col_tridiag,
                      10, rdata, 4, 4, 10);
std::cout << mysparse << std::endl;
```

will display the following output to the screen:

```
(1,1)    2
(2,1)    -1
(1,2)    -1
(2,2)    2
(3,2)    -1
```

```
(2,3)    -1
(3,3)    2
(4,3)    -1
(3,4)    -1
(4,4)    2
```

**static mxArray NewSparse(mwSize rowindex\_size, const mwIndex\* rowindex, mwSize colindex\_size, const mwIndex\* colindex, mwSize data\_size, const mxDouble\* rdata, mwSize nzmax)**

### Description

Creates real sparse matrix of type `double` with number of rows and columns inferred from input data.

The lengths of input row and column index and data arrays must all be the same or equal to 1. In the case where any of these arrays are equal to 1, the value is repeated through out the construction of the matrix.

If the same row/column pair occurs more than once, the data value assigned to that element is the sum of all values associated with that pair. The number of rows and columns in the created matrix are calculated from the input `rowindex` and `colindex` arrays as `num_rows = max{rowindex}`, `num_cols = max{colindex}`.

### Arguments

<code>mwSize rowindex_size</code>	Size of <code>rowindex</code> array
<code>mwIndex* rowindex</code>	Array of row indices of non-zero elements
<code>mwSize colindex_size</code>	Size of <code>colindex</code> array
<code>mwIndex* colindex</code>	Array of column indices of non-zero elements
<code>mwSize data_size</code>	Size of data array
<code>mxDouble* rdata</code>	Data associated with non-zero row and column indices
<code>mwSize nzmax</code>	Reserved storage for sparse matrix. If <code>nzmax</code> is zero, storage will be set to <code>max{rowindex_size, colindex_size, data_size}</code> .

### Example

In this example, we construct a sparse 4-by-4 identity matrix. The value of 1.0 is copied to each non-zero element defined by row and column index arrays:

```
double one = 1.0;
mwIndex row_diag[] = {1, 2, 3, 4};
mwIndex col_diag[] = {1, 2, 3, 4};

mxArray mysparse =
    mxArray::NewSparse(4, row_diag,
                      4, col_diag,
                      1, &one,
                      0);
std::cout << mysparse << std::endl;
```



```
(1,1)    1
(2,2)    1
(3,3)    1
(4,4)    1
```

**static mwArray NewSparse(mwSize rowindex\_size, const mwIndex\* rowindex, mwSize colindex\_size, const mwIndex\* colindex, mwSize data\_size, const mxDouble\* rdata, const mxDouble\* idata, mwSize num\_rows, mwSize num\_cols, mwSize nzmax)**

### Description

Creates complex sparse matrix of type double with specified number of rows and columns.

The lengths of input row and column index and data arrays must all be the same or equal to 1. In the case where any of these arrays are equal to 1, the value is repeated through out the construction of the matrix.

If the same row/column pair occurs more than once, the data value assigned to that element is the sum of all values associated with that pair. If any element of the rowindex or colindex array is greater than the specified values in num\_rows, num\_cols, respectively, then an exception is thrown.

### Arguments

mwSize rowindex_size	Size of rowindex array
mwIndex* rowindex	Array of row indices of non-zero elements
mwSize colindex_size	Size of colindex array
mwIndex* colindex	Array of column indices of non-zero elements
mwSize data_size	Size of data array
mxDouble* rdata	Real part of data associated with non-zero row and column indices
mxDouble* idata	Imaginary part of data associated with non-zero row and column indices
mwSize num_rows	Number of rows in matrix
mwSize num_cols	Number of columns in matrix
mwSize nzmax	Reserved storage for sparse matrix. If nzmax is zero, storage will be set to $\max\{\text{rowindex\_size}, \text{colindex\_size}, \text{data\_size}\}$ .

### Example

This example constructs a complex tridiagonal matrix:

```
double rdata[] =
    {2.0, -1.0, -1.0, 2.0, -1.0, -1.0, 2.0, -1.0, -1.0, 2.0};
double idata[] =
    {20.0, -10.0, -10.0, 20.0, -10.0, -10.0, 20.0, -10.0,
     -10.0, 20.0};

mwIndex row_tridiag[] =
    {1, 2, 1, 2, 3, 2, 3, 4, 3, 4};
mwIndex col_tridiag[] =
    {1, 1, 2, 2, 2, 3, 3, 3, 4, 4};
```

```

mwArray mysparse = mxArray::NewSparse(10, row_tridiag,
                                     10, col_tridiag,
                                     10, rdata,
                                     idata, 4, 4, 10);
std::cout << mysparse << std::endl;

```

It displays the following output to the screen:

```

(1,1)    2.0000 +20.0000i
(2,1)   -1.0000 -10.0000i
(1,2)   -1.0000 -10.0000i
(2,2)    2.0000 +20.0000i
(3,2)   -1.0000 -10.0000i
(2,3)   -1.0000 -10.0000i
(3,3)    2.0000 +20.0000i
(4,3)   -1.0000 -10.0000i
(3,4)   -1.0000 -10.0000i
(4,4)    2.0000 +20.0000i

```

**static mxArray NewSparse(mwSize rowindex\_size, const mwIndex\* rowindex, mwSize colindex\_size, const mwIndex\* colindex, mwSize data\_size, const mxDouble\* rdata, const mxDouble\* idata, mwSize nzmax)**

#### Description

Creates complex sparse matrix of type double with number of rows and columns inferred from input data.

The lengths of input row and column index and data arrays must all be the same or equal to 1. In the case where any of these arrays are equal to 1, the value is repeated through out the construction of the matrix.

If the same row/column pair occurs more than once, the data value assigned to that element is the sum of all values associated with that pair. The number of rows and columns in the created matrix are calculated from the input rowindex and colindex arrays as  $\text{num\_rows} = \max\{\text{rowindex}\}$ ,  $\text{num\_cols} = \max\{\text{colindex}\}$ .

#### Arguments

mwSize rowindex_size	Size of rowindex array
mwIndex* rowindex	Array of row indices of non-zero elements
mwSize colindex_size	Size of colindex array
mwIndex* colindex	Array of column indices of non-zero elements
mwSize data_size	Size of data array
mxDouble* rdata	Real part of data associated with non-zero row and column indices
mxDouble* idata	Imaginary part of data associated with non-zero row and column indices

<code>mwSize nzmax</code>	Reserved storage for sparse matrix. If <code>nzmax</code> is zero, storage will be set to <code>max{rowindex_size, colindex_size, data_size}</code> .
---------------------------	---

**Example**

This example constructs a complex matrix by inferring dimensions and storage allocation from the input data.

```
mwArray mysparse =
    mwArray::NewSparse(10, row_tridiag,
                      10, col_tridiag,
                      10, rdata, idata,
                      0);
std::cout << mysparse << std::endl;
```

```
(1,1)    2.0000 +20.0000i
(2,1)   -1.0000 -10.0000i
(1,2)   -1.0000 -10.0000i
(2,2)    2.0000 +20.0000i
(3,2)   -1.0000 -10.0000i
(2,3)   -1.0000 -10.0000i
(3,3)    2.0000 +20.0000i
(4,3)   -1.0000 -10.0000i
(3,4)   -1.0000 -10.0000i
(4,4)    2.0000 +20.0000i
```

**static mwArray NewSparse(mwSize rowindex\_size, const mwIndex\* rowindex, mwSize colindex\_size, const mwIndex\* colindex, mwSize data\_size, const mxLogical\* rdata, mwSize num\_rows, mwSize num\_cols, mwSize nzmax)**

**Description**

Creates logical sparse matrix with specified number of rows and columns.

The lengths of input row and column index and data arrays must all be the same or equal to 1. In the case where any of these arrays are equal to 1, the value is repeated throughout the construction of the matrix.

If the same row/column pair occurs more than once, the data value assigned to that element is the sum of all values associated with that pair. If any element of the `rowindex` or `colindex` array is greater than the specified values in `num_rows`, `num_cols`, respectively, then an exception is thrown.

**Arguments**

<code>mwSize rowindex_size</code>	Size of <code>rowindex</code> array
<code>mwIndex* rowindex</code>	Array of row indices of non-zero elements
<code>mwSize colindex_size</code>	Size of <code>colindex</code> array
<code>mwIndex* colindex</code>	Array of column indices of non-zero elements
<code>mwSize data_size</code>	Size of data array
<code>mxLogical* rdata</code>	Data associated with non-zero row and column indices

<code>mwSize num_rows</code>	Number of rows in matrix
<code>mwSize num_cols</code>	Number of columns in matrix
<code>mwSize nzmax</code>	Reserved storage for sparse matrix. If <code>nzmax</code> is zero, storage will be set to <code>max{rowindex_size, colindex_size, data_size}</code> .

**Example**

This example creates a sparse logical 4-by-4 tridiagonal matrix, assigning `true` to each non-zero value:

```

mxLogical one = true;
mwIndex row_tridiag[] =
    {1, 2, 1, 2, 3,
     2, 3, 4, 3, 4};
mwIndex col_tridiag[] =
    {1, 1, 2, 2, 2,
     3, 3, 3, 4, 4};

mwArray mysparse =
    mxArray::NewSparse(10, row_tridiag,
                      10, col_tridiag,
                      1, &one,
                      4, 4, 10);

std::cout << mysparse << std::endl;

(1,1)      1
(2,1)      1
(1,2)      1
(2,2)      1
(3,2)      1
(2,3)      1
(3,3)      1
(4,3)      1
(3,4)      1
(4,4)      1

```

**static mxArray NewSparse(mwSize rowindex\_size, const mwIndex\* rowindex, mwSize colindex\_size, const mwIndex\* colindex, mwSize data\_size, const mxLogical\* rdata, mwSize nzmax)**

**Description**

Creates logical sparse matrix with number of rows and columns inferred from input data.

The lengths of input row and column index and data arrays must all be the same or equal to 1. In the case where any of these arrays are equal to 1, the value is repeated through out the construction of the matrix.

The number of rows and columns in the created matrix are calculated form the input `rowindex` and `colindex` arrays as `num_rows = max {rowindex}`, `num_cols = max {colindex}`.

**Arguments**

mwSize rowindex_size	Size of rowindex array
mwIndex* rowindex	Array of row indices of non-zero elements
mwSize colindex_size	Size of colindex array
mwIndex* colindex	Array of column indices of non-zero elements
mwSize data_size	Size of data array
mxLogical* rdata	Data associated with non-zero row and column indices
mwSize nzmax	Reserved storage for sparse matrix. If nzmax is zero, storage will be set to $\max\{\text{rowindex\_size}, \text{colindex\_size}, \text{data\_size}\}$ .

**Example**

This example uses the data from the first example, but allows the number of rows, number of columns, and allocated storage to be calculated from the input data:

```
mwArray mysparse =
    mwArray::NewSparse(10, row_tridiag,
                      10, col_tridiag,
                      1, &one,
                      0);
std::cout << mysparse << std::endl;

(1,1)      1
(2,1)      1
(1,2)      1
(2,2)      1
(3,2)      1
(2,3)      1
(3,3)      1
(4,3)      1
(3,4)      1
(4,4)      1
```

**static mwArray NewSparse (mwSize num\_rows, mwSize num\_cols, mwSize nzmax, mxClassID mxID, mxComplexity cmplx = mxREAL)**

**Description**

Creates an empty sparse matrix. All elements in an empty sparse matrix are initially zero, and the amount of allocated storage for non-zero elements is specified by nzmax.

**Arguments**

mwSize num_rows	Number of rows in matrix
mwSize num_cols	Number of columns in matrix
mwSize nzmax	Reserved storage for sparse matrix

mxClassID mxID	Type of data to store in matrix. Currently, sparse matrices of type <code>double</code> precision and <code>logical</code> are supported. Pass <code>mxDOUBLE_CLASS</code> to create a <code>double</code> precision sparse matrix. Pass <code>mxLOGICAL_CLASS</code> to create a <code>logical</code> sparse matrix.
mxComplexity cplx	Complexity of matrix. Pass <code>mxCOMPLEX</code> to create a <code>complex</code> sparse matrix and <code>mxREAL</code> to create a <code>real</code> sparse matrix. This argument may be omitted, in which case the default complexity is <code>real</code> .

**Example**

This example constructs a real 3-by-3 empty sparse matrix of type `double` with reserved storage for 4 non-zero elements:

```
mwArray mysparse = mxArray::NewSparse
    (3, 3, 4, mxDOUBLE_CLASS);
std::cout << mysparse << std::endl;
```

All zero sparse: 3-by-3

**static double GetNaN()****Description**

Get value of NaN (Not-a-Number).

Call `mwArray::GetNaN` to return the value of NaN for your system. NaN is the IEEE arithmetic representation for Not-a-Number. Certain mathematical operations return NaN as a result, for example:

- 0.0/0.0
- Inf-Inf

The value of NaN is built in to the system; you cannot modify it.

**Example**

```
double x = mxArray::GetNaN();
```

**static double GetEps()****Description**

Returns the value of the MATLAB `eps` variable. This variable is the distance from 1.0 to the next largest floating-point number. Consequently, it is a measure of floating-point accuracy. The MATLAB `pinv` and `rank` functions use `eps` as a default tolerance.

**Example**

```
double x = mxArray::GetEps();
```

**static double GetInf()****Description**

Returns the value of the MATLAB internal `Inf` variable. `Inf` is a permanent variable representing IEEE arithmetic positive infinity. The value of `Inf` is built into the system; you cannot modify it.

Operations that return `Inf` include

- Division by 0. For example, `5/0` returns `Inf`.
- Operations resulting in overflow. For example, `exp(10000)` returns `Inf` because the result is too large to be represented on your machine.

**Example**

```
double x = mwArray::GetInf();
```

**static bool IsFinite(double x)****Description**

Determine whether or not a value is finite. A number is finite if it is greater than `-Inf` and less than `Inf`.

**Arguments**

<code>double x</code>	Value to test for finiteness
-----------------------	------------------------------

**Example**

```
bool x = mwArray::IsFinite(1.0);
```

**static bool IsInf(double x)****Description**

Determines whether or not a value is equal to infinity or minus infinity. MATLAB stores the value of infinity in a permanent variable named `Inf`, which represents IEEE arithmetic positive infinity. The value of the variable, `Inf`, is built into the system; you cannot modify it.

Operations that return infinity include

- Division by 0. For example, `5/0` returns infinity.
- Operations resulting in overflow. For example, `exp(10000)` returns infinity because the result is too large to be represented on your machine. If the value equals `NaN` (Not-a-Number), then `mXIsInf` returns `false`. In other words, `NaN` is not equal to infinity.

**Arguments**

<code>double x</code>	Value to test for infiniteness
-----------------------	--------------------------------

**Example**

```
bool x = mwArray::IsInf(1.0);
```

**static bool IsNaN(double x)****Description**

Determines whether or not the value is NaN. NaN is the IEEE arithmetic representation for Not-a-Number. NaN is obtained as a result of mathematically undefined operations such as

- 0.0/0.0
- Inf-Inf

The system understands a family of bit patterns as representing NaN. In other words, NaN is not a single value, rather it is a family of numbers that the MATLAB software (and other IEEE-compliant applications) use to represent an error condition or missing data.

**Arguments**

double x	Value to test for NaN
----------	-----------------------

**Example**

```
bool x = mxArray::IsNaN(1.0);
```

**Operators****mwArray operator()(mwIndex i1, mwIndex i2, mwIndex i3, ..., )****Description**

Fetches a single element at a specified index. The index is passed as a comma-separated list of 1-based indices. This operator is overloaded to support 1 through 32 indices. The valid number of indices that can be passed in is either 1 (single subscript indexing) or `NumberOfDimensions()` (multiple subscript indexing). In single subscript indexing the element at the specified 1-based offset is returned, accessing data in column-wise order. In multiple subscript indexing the index list is used to access the specified element. The valid range for indices is  $1 \leq \text{index} \leq \text{NumberOfElements}()$ , for single subscript indexing. For multiple subscript indexing, the  $i$ th index has the valid range:  $1 \leq \text{index}[i] \leq \text{GetDimensions}().\text{Get}(1, i)$ . An `mwException` is thrown if an invalid number of indices is passed in or if any index is out of bounds.

**Arguments**

mwIndex i1, mwIndex i2, mwIndex i3, ...,	Comma-separated list of input indices
--	---------------------------------------

**Example**

```
double data[4] = {1.0, 2.0, 3.0, 4.0};
double x;
mwArray a(2, 2, mxDOUBLE_CLASS);
a.SetData(data, 4);
x = a(1,1);
x = a(1,2);
x = a(2,2);
```



**mwArray operator()(const char\* name, mwIndex i1, mwIndex i2, mwIndex i3, ..., )****Description**

Fetches a single element at a specified field name and index. This method may only be called on an array that is of type `mxSTRUCT_CLASS`. An `mwException` is thrown if the underlying array is not a `struct` array. The field name passed must be a valid field name in the `struct` array. The index is passed by first passing the number of indices, followed by an array of 1-based indices. This operator is overloaded to support 1 through 32 indices. The valid number of indices that can be passed in is either 1 (single subscript indexing) or `NumberOfDimensions()` (multiple subscript indexing). In single subscript indexing the element at the specified 1-based offset is returned, accessing data in column-wise order. In multiple subscript indexing the index list is used to access the specified element. The valid range for indices is `1 <= index <= NumberOfElements()`, for single subscript indexing. For multiple subscript indexing, the *i*th index has the valid range: `1 <= index[i] <= GetDimensions().Get(1, i)`. An `mwException` is thrown if an invalid number of indices is passed in or if any index is out of bounds.

**Arguments**

<code>char* name</code>	Null terminated string containing the field name to get
<code>mwIndex i1, mwIndex i2, mwIndex i3, ...,</code>	Comma-separated list of input indices

**Example**

```
const char* fields[] = {"a", "b", "c"};
int index[2] = {1, 1};
mwArray a(1, 1, 3, fields);
mwArray b = a("a", 1, 1);
mwArray b = a("b", 1, 1);
```

**mwArray& operator=(const <type>& x)****Description**

Sets a single scalar value. This operator is overloaded for all numeric and logical types.

**Arguments**

<code>const &lt;type&gt;&amp; x</code>	Value to assign
--	-----------------

**Example**

```
mwArray a(2, 2, mxDOUBLE_CLASS);
a(1,1) = 1.0;
a(1,2) = 2.0;
a(2,1) = 3.0;
a(2,2) = 4.0;
```

**const mwArray operator()(mwIndex i1, mwIndex i2, mwIndex i3, ..., ) const****Description**

Fetches a single scalar value. This operator is overloaded for all numeric and logical types.

**Arguments**

<code>mwIndex i1, mwIndex i2, mwIndex i3, ...,</code>	Comma-separated list of input indices
---	---------------------------------------

**Example**

```
double data[4] = {1.0, 2.0, 3.0, 4.0};
double x;
mwArray a(2, 2, mxDOUBLE_CLASS);
a.SetData(data, 4);
x = (double)a(1,1);
x = (double)a(1,2);
x = (double)a(2,1);
x = (double)a(2,2);
```

**std::ostream::operator<<(const mxArray &)****Description**

Write mxArray to output stream. The output has the same format as the output when a variable's name is typed at the MATLAB command prompt. See ToString().

## Version History

**Introduced in R2013b**

# C++ MATLAB Data API

---

## matlab::cpplib::initMATLABApplication

Start the MATLAB Runtime and initialize its application state

### Description

```
std::shared_ptr<MATLABApplication>
initMATLABApplication(matlab::cpplib::MATLABApplicationMode mode, const
std::vector<std::u16string>& options = std::vector<std::u16string>())
```

`matlab.cpplib.initMATLABApplication` accepts as input `mode` and an optional array of startup options. It returns a shared pointer to a `MATLABApplication` object. The shared pointer is passed to the function `matlab::cpplib::initMATLABLibrary`, which returns a unique pointer to a user written library. This unique pointer is then used to call MATLAB functions from the library

A process should call this method only once.

### Parameters

<code>MATLABApplicationMode mode</code>	Mode in which to start application: <ul style="list-style-type: none"> <li><code>MATLABApplicationMode::IN_PROCESS</code></li> <li><code>MATLABApplicationMode::OUT_OF_PROCESS</code></li> </ul>
<code>const std::vector&lt;std::u16string&gt;&amp; options</code>	Start up options used to start a MATLAB Runtime. They include: <ul style="list-style-type: none"> <li><code>-nodisplay</code>: Starts the MATLAB Runtime without display functionality on Linux.</li> <li><code>-nojvm</code>: Disables the Java Virtual Machine, which is enabled by default.</li> <li><code>-logfile filepath</code>: Writes to the log file with path <i>filepath</i>. <code>-logfile</code> and <i>filepath</i> must be specified as separate consecutive arguments.</li> </ul>

### Return Value

<code>std::shared_ptr&lt;MATLABApplication&gt;</code>	Pointer to a <code>MATLABApplication</code> object that encapsulates the application state.
---	---

### Exceptions

<code>matlab::cpplib::ApplicationLaunchError</code>	The function failed to start.
---	-------------------------------

## Examples

### Start MATLAB Runtime In-Process, with Default Runtime Options

```
std::shared_ptr<MATLABApplication> appPtr = initMATLABApplication(MATLABApplicationMode::IN_PROCE
```

### Start MATLAB Runtime Out-Of-Process, Without a Java Virtual Machine

```
std::vector<std::string> opts = {"-nojvm"};  
std::shared_ptr<MATLABApplication> appPtr = initMATLABApplication(MATLABApplicationMode::OUT_OF_P
```

### Start MATLAB Runtime In-Process, and Generate a Log File

```
std::vector<std::u16string> opts = {u"-logfile",  
                                     u"C:\\somepath\\matlab_app.log"};  
std::shared_ptr<MATLABApplication> appPtr = initMATLABApplication(MATLABApplicationMode::IN_PROCE
```

## Version History

Introduced in R2018a

### See Also

matlab::cpplib::runMain | matlab::cpplib::convertUTF8StringToUTF16String |  
matlab::cpplib::convertUTF16StringToUTF8String |  
matlab::cpplib::initMATLABLibrary | matlab::cpplib::initMATLABLibraryAsync |  
matlab::cpplib::MATLABLibrary::feval |  
matlab::cpplib::MATLABLibrary::fevalAsync |  
matlab::cpplib::MATLABLibrary::waitForFiguresToClose

## matlab::cpplib::runMain

Execute a function with its input arguments within the main function

### Description

```
int runMain(std::function<int(std::shared_ptr<MatlabApplication> func, int,
const char**)>std::shared_ptr<MatlabApplication>&& app, int argc, const char
**argv);
```

Execute a function with its input arguments within the main function. `matlab.cpplib.runMain` accepts as input the function you want to execute, an instance of `MATLABApplication`, and the inputs to the function you want to execute. It returns as output a code indicating the success or failure of execution.

This function can be used on any platform to separate the logic of the primary function from that of `main()`. On macOS, it also fulfills the requirements of the Cocoa API

### Parameters

<code>std::function&lt;int(std::shared_ptr&lt;MATLABApplication&gt;, int, const char**)&gt; func</code>	A <code>std::function</code> instance that takes three parameters (namely, a pointer to a <code>MATLABApplication</code> object, an <code>int</code> representing the number of input arguments, and a <code>const char**</code> representing the input arguments themselves) and returns an <code>int</code> .
<code>std::shared_ptr&lt;MATLABApplication&gt;&amp;&amp; app</code>	Instance of <code>MATLABApplication</code> , passed as <code>rvalue</code> .
<code>int argc</code>	Number of input arguments from the command line.
<code>const char **argv</code>	Input arguments array.

### Return Value

<code>int</code>	Return code indicating success (by convention: 0), or failure (by convention, a non-zero number).
------------------	---

### Examples

#### Move the MATLABApplication Object into runMain and Terminate It

```
int myMainFunc(std::shared_ptr<mc::MATLABApplication> app,
const int argc, const char * argv[])
{
    try {
        // initialize library, call feval, etc.
    } catch(const std::exception & exc) {
        std::cerr << exc.what() << std::endl;
        return -1;
    }
    return 0; // no error
}
```

```
int main(const int argc, const char * argv[])
{
    std::vector<std::u16string> options ;
    auto matlabApplication = mc::initMATLABApplication(
        mc::MATLABApplicationMode::IN_PROCESS,options);
    return mc::runMain(myMainFunc, std::move(matlabApplication), argc, argv);
}
```

## Version History

Introduced in R2018a

### See Also

matlab::cpplib::initMATLABApplication |  
matlab::cpplib::convertUTF8StringToUTF16String |  
matlab::cpplib::convertUTF16StringToUTF8String |  
matlab::cpplib::initMATLABLibrary | matlab::cpplib::initMATLABLibraryAsync |  
matlab::cpplib::MATLABLibrary::feval |  
matlab::cpplib::MATLABLibrary::fevalAsync |  
matlab::cpplib::MATLABLibrary::waitForFiguresToClose

## matlab::cpplib::convertUTF8StringToUTF16String

Convert UTF-8 string to UTF-16 string

### Description

```
std::u16string & ustr convertUTF8StringToUTF16String(const std::string & str)
```

Convert a UTF-8 string (ASCII or Unicode®) to a UTF-16 string. Use this function to convert ASCII strings into the form required to represent start-up options (passed to `initMATLABApplication`), or function names or `matlab::data::array`.

Prefixing `u` to a literal `char * string` is a more concise alternative that achieves the same effect as `convertUTF8StringToUTF16String` when a literal string is passed as a parameter. For example, you could write `initMATLABLibrary(app, u"mylib");` rather than the lengthier `initMATLABLibrary(app, convertUTF8StringToUTF16String("mylib"));` and get the same results.

---

**Note** Prefixing `u` is not supported by Visual C++® 2013.

---

### Parameters

```
const std::string & A UTF-8 (possibly ASCII) string.  
str
```

### Return Value

```
std::u16string A UTF-16-encoded string.
```

### Exceptions

```
std::range_err Input is not a valid UTF-8 string.  
or
```

### Examples

#### Convert UTF-8 String to UTF-16 String

```
auto app = initMATLABApplication(MATLABApplicationMode::IN_PROCESS);  
const char * libName = getLibNameFromConfigFile(); // imaginary user-defined function  
auto mylib = initMATLABLibrary(app, convertUTF8StringToUTF16String(libName));
```

### Version History

Introduced in R2018a



**See Also**

matlab::cpplib::convertUTF16StringToUTF8String |  
matlab::cpplib::initMATLABApplication | matlab::cpplib::runMain |  
matlab::cpplib::initMATLABLibrary | matlab::cpplib::initMATLABLibraryAsync |  
matlab::cpplib::MATLABLibrary::feval |  
matlab::cpplib::MATLABLibrary::fevalAsync |  
matlab::cpplib::MATLABLibrary::waitForFiguresToClose

## matlab::cpplib::convertUTF16StringToUTF8String

Convert UTF-16 string to UTF-8 string

### Description

```
std::string & str convertUTF16StringToUTF8String(const std::u16string & ustr)
```

Convert a UTF-16 string to a UTF-8 string. Since ASCII is a subset of UTF-8 encoding, the output is ASCII content as long as no non-ASCII characters are present in the input.

### Parameters

```
const std::u16string A UTF-16 string.  
& ustr
```

### Return Value

```
std::string A UTF-8 string.
```

### Exceptions

```
std::range_err Input is not valid UTF-16 string.  
or
```

### Examples

#### Convert a UTF-16 String to UTF-8 String

```
auto app = initMATLABApplication(MATLABApplicationMode::OUT_OF_PROCESS);  
auto mylib = initMATLABLibrary(app, convertUTF8StringToUTF16String("mylib"));  
std::u16string ustr = mylib->feval<std::u16string>("get_const_str");  
std::string str = convertUTF16StringToUTF8String(ustr);
```

## Version History

Introduced in R2017b

### See Also

```
matlab::cpplib::convertUTF8StringToUTF16String |  
matlab::cpplib::initMATLABApplication | matlab::cpplib::runMain |  
matlab::cpplib::initMATLABLibrary | matlab::cpplib::initMATLABLibraryAsync |  
matlab::cpplib::MATLABLibrary::feval |  
matlab::cpplib::MATLABLibrary::fevalAsync |  
matlab::cpplib::MATLABLibrary::waitForFiguresToClose
```

# matlab::cpplib::initMATLABLibrary

Initialize a library of MATLAB functions packaged in a deployable archive file

## Description

```
std::unique_ptr<MATLABLibrary>
initMATLABLibrary(std::shared_ptr<MATLABApplication> application, const
std::u16string & ctfPath)
```

Initialize a library of MATLAB functions packaged in a deployable archive (CTF) file, and return a unique pointer to the library. As parameters, it takes a shared pointer to a `MATLABApplication` instance and a path to the CTF.

The path to the deployable archive is either relative or absolute. If the path is relative, the following paths are prepended in the order specified below until the file is found or all possibilities are exhausted.

- the value of the environment variable `CPPSHARED_BASE_CTF_PATH`, if defined
- the working folder
- the folder where the executable is located
- on Mac: the folder three levels above the folder where the executable is located (for example, if the executable is `generic_interface/foo_generic.app/Contents/MacOS/foo`, the folder used is `generic_interface`)

If the library is found, it is initialized and a pointer to it is returned. Otherwise, an exception is thrown.

## Parameters

```
std::shared_ptr<MATLABApplication> application  Pointer to a MATLABApplication object returned from
initMATLABApplication.
const std::u16string & ctfPath  Path (relative or absolute) to archive.
```

## Return Value

```
std::unique_ptr<MATLABLibrary>  Pointer to a MATLABLibrary object that is used to call functions from the
library, feval etc.
```

## Exceptions

```
matlab::cpplib::LibNotFound  No library with the given name is found on the shared library path.
matlab::cpplib::LibInitErr  Library cannot be initialized.
```

## Examples

### Initialize MATLABLibrary

```
std::vector<std::u16string> opts = {u"-nojvm"};  
auto matlabPtr = initMATLABApplication(MATLABApplicationMode::IN_PROCESS, opts);  
auto libAstro = initMATLABLibrary(matlabPtr, convertUTF8StringToUTF16String("astro.ctf"));
```

## Version History

Introduced in R2018a

### See Also

```
matlab::cpplib::convertUTF8StringToUTF16String |  
matlab::cpplib::convertUTF16StringToUTF8String |  
matlab::cpplib::initMATLABApplication | matlab::cpplib::runMain |  
matlab::cpplib::initMATLABLibraryAsync | matlab::cpplib::MATLABLibrary::feval |  
matlab::cpplib::MATLABLibrary::fevalAsync |  
matlab::cpplib::MATLABLibrary::waitForFiguresToClose
```

# matlab::cpplib::initMATLABLibraryAsync

Initialize a library of MATLAB function asynchronously

## Description

```
FutureResult<std::shared_ptr<MATLABLib>>
initMATLABLibraryAsync(MATLABApplication & application, const std::u16string
& ctfPath)
```

Initialize a library of MATLAB function asynchronously, to obtain a pointer to a freshly initialized C++ shared library once initialization is complete.

## Parameters

`MATLABApplication & application` MATLAB Application object returned from `initMATLABApplication`.

`const std::u16string & ctfPath` Name of library. If path is omitted, it is assumed to be in the current folder. For information on how to use `ctfPath`, see `matlab::cpplib::initMATLABLibrary`.

## Return Value

`FutureResult<std::shared_ptr<MATLABLib>>` A `std::future` from which the status of initialization process, or a library pointer (once initialization is complete) can be obtained.

## Exceptions

`matlab::cpplib::LibNotFound` No library with the given name found on the shared library path.

`matlab::cpplib::LibInitErr` Library cannot be initialized.

## Examples

### Initialize MATLABLibrary Asynchronously, and Wait Until It Initializes

```
auto future = mc::initMatlabLibraryAsync(matlabApplication,
    mc::convertUTF8StringToUTF16String("libdoubleasync.ctf"));
if (!future.valid()) {
    throw std::future_error(std::future_errc::no_state);
}
std::future_status status;
do {
    status = future.wait_for(std::chrono::milliseconds(200));
    if (status == std::future_status::timeout) {
        std::cout << "Library initialization is in progress.\n";
    } else if (status == std::future_status::ready) {
        std::cout << "Library initialization has completed.\n";
    }
}
```

```
    }  
    std::this_thread::sleep_for(std::chrono::seconds(1));  
} while (status != std::future_status::ready);  
auto lib = future.get();
```

## **Version History**

**Introduced in R2018a**

### **See Also**

`matlab::cpplib::convertUTF8StringToUTF16String` |  
`matlab::cpplib::convertUTF16StringToUTF8String` |  
`matlab::cpplib::initMATLABApplication` | `matlab::cpplib::runMain` |  
`matlab::cpplib::initMATLABLibrary` | `matlab::cpplib::MATLABLibrary::feval` |  
`matlab::cpplib::MATLABLibrary::fevalAsync` |  
`matlab::cpplib::MATLABLibrary::waitForFiguresToClose`

# matlab::cpplib::MATLABLibrary::feval

Execute a MATLAB function from a deployable archive

## Description

**Execute a function with 1 output MATLAB Data Array argument; 1 input MATLAB Data Array argument**

*function name as u16string*

```
matlab::data::Array feval(const std::u16string &function, const
matlab::data::Array &arg, const std::shared_ptr<StreamBuffer> &output =
std::shared_ptr<StreamBuffer>(), const std::shared_ptr<StreamBuffer> &error =
std::shared_ptr<StreamBuffer>())
```

*function name as string*

```
matlab::data::Array feval(const std::string &function, const
matlab::data::Array &arg, const std::shared_ptr<StreamBuffer> &output =
std::shared_ptr<StreamBuffer>(), const std::shared_ptr<StreamBuffer> &error =
std::shared_ptr<StreamBuffer>())
```

**Execute a function with 1 output MATLAB Data Array argument; 0, 2, or more input MATLAB Data Array arguments**

*function name as u16string*

```
matlab::data::Array feval(const std::u16string &function, const
std::vector<matlab::data::Array> &args, const std::shared_ptr<StreamBuffer>
&output = std::shared_ptr<StreamBuffer>(), const
std::shared_ptr<StreamBuffer> &error = std::shared_ptr<StreamBuffer>())
```

*function name as string*

```
matlab::data::Array feval(const std::string &function, const
std::vector<matlab::data::Array> &args, const std::shared_ptr<StreamBuffer>
&output = std::shared_ptr<StreamBuffer>(), const
std::shared_ptr<StreamBuffer> &error = std::shared_ptr<StreamBuffer>())
```

**Execute a function with 0, 2, or more output MATLAB Data Array arguments; any number of input MATLAB Data Array arguments**

*function name as u16string*

```
std::vector<matlab::data::Array> feval(const std::u16string &function, const
size_t nlhs, const std::vector<matlab::data::Array> &args, const
std::shared_ptr<StreamBuffer> &output = std::shared_ptr<StreamBuffer>(),
const std::shared_ptr<StreamBuffer> &error = std::shared_ptr<StreamBuffer>())
```

*function name as string*

```
std::vector<matlab::data::Array> feval(const std::string &function, const
size_t nlhs, const std::vector<matlab::data::Array> &args, const
std::shared_ptr<StreamBuffer> &output = std::shared_ptr<StreamBuffer>(),
const std::shared_ptr<StreamBuffer> &error = std::shared_ptr<StreamBuffer>())
```

### Execute a function with native input and output arguments

*function name as u16string*

```
template<class ReturnT, typename...RhsArgs> ReturnT feval(const
std::u16string &function, RhsArgs&&... rhsArgs)
```

*function name as string*

```
template<class ReturnT, typename...RhsArgs> ReturnT feval(const
std::string &function, RhsArgs&&... rhsArgs)
```

### Execute a function with native input and output arguments, with output redirection

*function name as u16string*

```
template<class ReturnT, typename...RhsArgs> ReturnT feval(const
std::u16string &function, const std::shared_ptr<StreamBuffer> &output, const
std::shared_ptr<StreamBuffer> &error, RhsArgs&&... rhsArgs)
```

*function name as string*

```
template<class ReturnT, typename...RhsArgs> ReturnT feval(const
std::string &function, const std::shared_ptr<StreamBuffer> &output, const
std::shared_ptr<StreamBuffer> &error, RhsArgs&&... rhsArgs)
```

Call a packaged MATLAB function within a C++ shared library:

- Without redirection of standard output or standard error
- With redirection of standard output
- With redirection of standard output and standard error

LhsItem	One of the following: <ul style="list-style-type: none"> <li>• Native scalar</li> <li>• <code>std::vector</code> of a native type</li> <li>• <code>matlab::data::Array</code></li> <li>• <code>std::tuple</code> of any combination of any of the previously mentioned possibilities.</li> </ul>
RhsArgs	A sequence of zero or more arguments which are one of the following: <ul style="list-style-type: none"> <li>• Native scalar</li> <li>• <code>std::vector</code> of a native type</li> <li>• <code>matlab::data::Array</code></li> </ul>
StreamBuffer	<code>std::basic_streambuf&lt;char16_t&gt;</code>

`MATLABLibrary::feval` calls a packaged MATLAB function within a C++ shared library and passes the name of the function, followed by the arguments. If the specified function cannot be found in the



library, an exception is thrown. By default, the function returns either a single `matlab::data::Array` object (if one output argument is expected) or a vector of `matlab::data::Array` objects (if zero or multiple output arguments are expected). In the former case, the vector is empty. By specifying a template argument, you can specify an alternative return type, which can be a primitive type, or a vector of primitive types, or a tuple of multiple instances of either.

Supported native types:

- `bool`
- `int8_t`
- `int16_t`
- `int32_t`
- `int64_t`
- `uint8_t`
- `uint16_t`
- `uint32_t`
- `uint64_t`
- `float`
- `double`
- `std::string`
- `std::u16string`
- `std::complex<T>` where T is one of the numeric types.
- Native C++ data passed as input is converted into the corresponding MATLAB types.
- `std::vector` is converted into a column array in MATLAB.
- The result of a MATLAB function is converted into the expected C++ data type if there is no loss of range.
- Otherwise, an exception is thrown.

## Parameters

<code>const std::u16string &amp;function</code>	The name of a compiled MATLAB function to be evaluated specified either as <code>u16string</code> or <code>string</code> .
<code>const std::string &amp;function</code>	
<code>const size_t nlhs</code>	The number of return values.
<code>const std::vector&lt;matlab::data::Array&gt;&amp; args</code>	Arguments used by the MATLAB function when more than one is specified.
<code>const matlab::data::Array&gt;&amp; arg</code>	Argument used by the MATLAB function with single input.

<code>const RhsArgs&amp; rhsArgs</code>	<p>Template parameter pack consisting of a sequence of zero or more arguments, each of which is one of the following:</p> <ul style="list-style-type: none"> <li>• a bare native type (see list of supported native types)</li> <li>• a <code>std::vector</code> of a bare native type</li> <li>• a <code>matlab::data::Array</code></li> </ul>
<code>const std::shared_ptr&lt;StreamBuffer&gt;&amp; output</code>	String buffer used to store the standard output from the MATLAB function.
<code>const std::shared_ptr&lt;StreamBuffer&gt;&amp; error</code>	String buffer used to store error output from the MATLAB function.

## Return Value

Zero or one of the following, or a tuple of any combination of them:

A native scalar type  
`std::vector`  
`matlab::data::Array`

## Exceptions

<code>matlab::cpplib::CanceledException</code>	The MATLAB function is canceled.
<code>matlab::cpplib::InterruptedException</code>	The MATLAB function is interrupted.
<code>matlab::cpplib::MATLABNotAvailableError</code>	The MATLAB session is not available.
<code>matlab::cpplib::MATLABSyntaxError</code>	The MATLAB function returned a syntax error.
<code>matlab::cpplib::MATLABExecutionError</code>	The function returns a MATLAB Runtime error.
<code>matlab::cpplib::TypeConversionError</code>	The result of a MATLAB function cannot be converted into a user-specific type.

## Examples

### Execute a User-Written MATLAB Function `mysqrt` in a C++ Shared Library

```
// This example assumes that mysqrt is a packaged user-written function that
// calls MATLAB's sqrt, which returns the square root of each element in
// the array that is passed to it.

auto matlabPtr = initMATLABApplication(MATLABApplicationMode::IN_PROCESS, opts);
auto libPtr = initMATLABLibrary(*matlabPtr, u"mylib.ctf");

// Initialize a matlab::data::TypedArray with three elements.
matlab::data::TypedArray<double> doubles = factory.createArray<double>({1.0, 4.0, 9.0});

// Retrieve the result of the mysqrt call. Since the output
// argument is a matlab::data::Array, feval does not require any template
```

```
// arguments.
matlab::data::Array mda = libPtr->feval(u"mysqrt", doubles);

// Now we retrieve the first element of that matlab::data::Array.
double d1 = mda[0];
std::assert(d1 == 1.0, "unexpected value");

// Pass a native type (a double) directly to mysqrt. Specify that you want
// a double (rather than a matlab::data::Array) as the return type.
double d2 = libPtr->feval<double>(u"mysqrt", 4.0);
std::assert(d2 == 2.0, "unexpected value");
```

## Version History

Introduced in R2018a

### See Also

matlab::cpplib::convertUTF8StringToUTF16String |  
matlab::cpplib::convertUTF16StringToUTF8String |  
matlab::cpplib::initMATLABApplication | matlab::cpplib::runMain |  
matlab::cpplib::initMATLABLibrary | matlab::cpplib::initMATLABLibraryAsync |  
matlab::cpplib::MATLABLibrary::fevalAsync |  
matlab::cpplib::MATLABLibrary::waitForFiguresToClose

## matlab::cpplib::MATLABLibrary::fevalAsync

Execute a MATLAB function from a deployable archive asynchronously

### Description

**Execute a function with 1 output MATLAB Data Array argument and 1 input MATLAB Data Array argument**

*function name as u16string*

```
FutureResult<matlab::data::Array> fevalAsync(const std::u16string &function,
const matlab::data::Array &arg, const std::shared_ptr<StreamBuffer> &output =
std::shared_ptr<StreamBuffer>(), const std::shared_ptr<StreamBuffer> &error =
std::shared_ptr<StreamBuffer>())
```

*function name as string*

```
FutureResult<matlab::data::Array> fevalAsync(const std::string &function,
const matlab::data::Array &arg, const std::shared_ptr<StreamBuffer> &output =
std::shared_ptr<StreamBuffer>(), const std::shared_ptr<StreamBuffer> &error =
std::shared_ptr<StreamBuffer>())
```

**Execute a function with 1 output MATLAB Data Array argument and any number of input MATLAB Data Array arguments**

*function name as u16string*

```
FutureResult<matlab::data::Array> fevalAsync(const std::u16string &function,
const std::vector<matlab::data::Array> &args, const
std::shared_ptr<StreamBuffer> &output = std::shared_ptr<StreamBuffer>(),
const std::shared_ptr<StreamBuffer> &error = std::shared_ptr<StreamBuffer>())
```

*function name as string*

```
FutureResult<matlab::data::Array> fevalAsync(const std::string &function,
const std::vector<matlab::data::Array> &args, const
std::shared_ptr<StreamBuffer> &output = std::shared_ptr<StreamBuffer>(),
const std::shared_ptr<StreamBuffer> &error = std::shared_ptr<StreamBuffer>())
```

**Execute a function with any number of output MATLAB Data Array arguments and any number of input MATLAB Data Array arguments**

*function name as u16string*

```
FutureResult<std::vector<matlab::data::Array>> fevalAsync(const
std::u16string &function, const size_t nlhs, const
std::vector<matlab::data::Array> &args, const std::shared_ptr<StreamBuffer>
&output = std::shared_ptr<StreamBuffer>(), const
std::shared_ptr<StreamBuffer> &error = std::shared_ptr<StreamBuffer>())
```

*function name as string*

```
FutureResult<std::vector<matlab::data::Array>> fevalAsync(const std::string
&function, const size_t nlhs, const std::vector<matlab::data::Array> &args,
const std::shared_ptr<StreamBuffer> &output =
std::shared_ptr<StreamBuffer>(), const std::shared_ptr<StreamBuffer> &error =
std::shared_ptr<StreamBuffer>())
```

### Execute a function with native scalar input and output arguments

*function name as ul6string*

```
template<class ReturnT, typename...RhsArgs>
```

```
FutureResult<ReturnT> fevalAsync(const std::ul6string &function,
RhsArgs&&... rhsArgs)
```

*function name as string*

```
template<class ReturnT, typename...RhsArgs>
```

```
FutureResult<ReturnT> fevalAsync(const std::string &function, RhsArgs&&...
rhsArgs)
```

### Execute a function with native scalar input and output arguments, with output redirection

*function name as ul6string*

```
template<class ReturnT, typename...RhsArgs>
```

```
FutureResult<ReturnT> fevalAsync(const std::ul6string &function, const
std::shared_ptr<StreamBuffer> &output, const std::shared_ptr<StreamBuffer>
&error, RhsArgs&&... rhsArgs)
```

*function name as string*

```
template<class ReturnT, typename...RhsArgs>
```

```
FutureResult<ReturnT> fevalAsync(const std::string &function, const
std::shared_ptr<StreamBuffer> &output, const std::shared_ptr<StreamBuffer>
&error, RhsArgs&&... rhsArgs)
```

Call a packaged MATLAB function within a C++ shared library asynchronously:

- Without redirection of standard output or standard error:
- With redirection of standard output:
- With redirection of standard output and standard error:

where,

LhsItem	native scalar
RhsArgs	A sequence of one or more native scalars.
StreamBuffer	std::basic_streambuf<char16_t>

It passes the name of the function, followed by the arguments. If the specified function cannot be found in the library, an exception is thrown.

Supported native types:

- `bool`
- `int8_t`
- `int16_t`
- `int32_t`
- `int64_t`
- `uint8_t`
- `uint16_t`
- `uint32_t`
- `uint64_t`
- `float`
- `double`
- `std::string`
- `std::u16string`
- `std::complex<T>` where T is one of the numeric types.

## Parameters

<code>const std::u16string &amp;function</code>	The name of a compiled MATLAB function to be evaluated specified either as <code>u16string</code> or <code>string</code> .
<code>const std::string &amp;function</code>	
<code>const size_t nlhs</code>	The number of return values.
<code>const std::vector&lt;matlab::data::Array&gt; args</code>	Arguments used by the MATLAB function.
<code>const matlab::data::Array arg</code>	Argument used by the MATLAB function with single input.
<code>const RhsArgs&amp; rhsArgs</code>	Template parameter pack consisting of a sequence of one or more arguments, each of which is a native scalar.
<code>const std::shared_ptr&lt;StreamBuffer&gt;&amp; output</code>	String buffer used to store the standard output from the MATLAB function.
<code>const std::shared_ptr&lt;StreamBuffer&gt;&amp; error</code>	String buffer used to store error output from the MATLAB function.

## Return Value

`FutureResult` Takes any of the permissible types for `LhsItem`.

## Exceptions

<code>matlab::cpplib::CanceledException</code>	The MATLAB function is canceled.
<code>matlab::cpplib::InterruptedException</code>	The MATLAB function is interrupted.

matlab::cpplib::MATLABNotAvailableError	The MATLAB session is not available.
matlab::cpplib::MATLABSyntaxError	The MATLAB function returned a syntax error.
matlab::cpplib::MATLABExecutionError	The function returns a MATLAB error.
matlab::cpplib::TypeConversionError	The result of a MATLAB function cannot be converted into a user-specific type.

## Examples

### Execute a User-Written MATLAB Function `repeatdouble` in a C++ Shared Library Asynchronously

```

/ Call the function repeatdouble, which iteratively continues to
// double a number, printing out results along the way. The
// (optional) second and third parameters determine, respectively, how
// many iterations should be performed and how many seconds should
// elapse between operations. We call the function asynchronously,
// then call it again (also asynchronously) with a different base
// number before all the iterations from the first call have completed.

// We pass the arguments to the function as C++ native types (namely
// doubles) rather than a md::Array. The return type is also a native
// type (a double), so fevalAsync and the FutureResult need to take
// "double" as a template parameter.
mc::FutureResult<double> futureA = lib->fevalAsync<double>(
    mc::convertUTF8StringToUTF16String("repeatdouble"), 10.0, 3.0, 0.5);
mc::FutureResult<double> futureB = lib->fevalAsync<double>(
    mc::convertUTF8StringToUTF16String("repeatdouble"), 11.0, 3.0, 0.5);

// Get the result once it's ready.
double resultA = futureA.get();
double resultB = futureB.get();

```

## Version History

Introduced in R2018a

### See Also

```

matlab::cpplib::convertUTF8StringToUTF16String |
matlab::cpplib::convertUTF16StringToUTF8String |
matlab::cpplib::initMATLABApplication | matlab::cpplib::runMain |
matlab::cpplib::initMATLABLibrary | matlab::cpplib::initMATLABLibraryAsync |
matlab::cpplib::MATLABLibrary::feval |
matlab::cpplib::MATLABLibrary::waitForFiguresToClose

```

# matlab::cpplib::MATLABLibrary::waitForFiguresToClose

Wait for all figures to close

## Description

`matlab::cpplib::MATLABLibrary::waitForFiguresToClose` method pauses until all figures in a library have been closed.

## Version History

Introduced in R2018a

## See Also

`matlab::cpplib::convertUTF8StringToUTF16String` |  
`matlab::cpplib::convertUTF16StringToUTF8String` |  
`matlab::cpplib::initMATLABApplication` | `matlab::cpplib::runMain` |  
`matlab::cpplib::initMATLABLibrary` | `matlab::cpplib::initMATLABLibraryAsync` |  
`matlab::cpplib::MATLABLibrary::feval` |  
`matlab::cpplib::MATLABLibrary::fevalAsync`



# **Workflow: C++ Shared Library using MATLAB Data API**

---

## Integrate C++ Shared Libraries with MATLAB Data API

### Workflow to Create C++ Shared Library using Generic Interface

To create a C++ shared library that uses the MATLAB Data API:

- 1 Package your MATLAB code into an archive (.ctf) file using the **Library Compiler** app or the `compiler.build.cppSharedLibrary` function. Select the MATLAB Data interface.
- 2 Write C++ code using the generic interface.
- 3 Link the driver code against header files provided with MATLAB Runtime.
- 4 Run your application.

For an example of this workflow, see “Generate a C++ MATLAB Data API Shared Library and Build a C++ Application”.

### Write C++ Code using Generic Interface

The basic workflow for using the generic interface for C++ shared libraries is as follows:

- Call the free function `initMATLABApplication`, which optionally takes a vector of run time options such as `-nojvm` and `-logfile`. The function returns a `shared_ptr`.
- Initialize a `matlab::data::ArrayFactory`, which you use to produce `matlab::data::Array` objects that you pass into function calls.
- For each library that you initialize, call `initMATLABLibrary`, which takes two parameters:
  - Copy of the `shared_ptr` that was returned by `initMATLABApplication`
  - Path to the archive (.ctf file)
- To call a function in an initialized library, call `feval` or `fevalAsync` on the `unique_ptr` that was returned by `initMATLABLibrary`. There are several overloaded versions of each. They all take the name of the MATLAB function as the first parameter. However, these differ in terms of whether they accept and return single `matlab::data::Array` objects, arrays of `matlab::data::Array`, or native types. The forms that return a native type must take the type as a template parameter.
- To terminate a library, either call `reset` on its `unique_ptr`, or allow it to go out of scope.
- To terminate the application, either call `reset` on its `shared_ptr`, or allow it to go out of scope. It does not terminate until all the libraries created underneath it have been terminated or gone out of scope.

For example code that uses the C++ MATLAB Data Array API, see `matrix_mda.cpp` located in `matlabroot\extern\examples\compilersdk\c_cpp\matrix`.

#### `matrix_mda.cpp`

```
/*=====
 *
 * MATRIX_MDA.CPP
 * Sample driver code that uses the generic interface
 * (introduced in R2018a) and MATLAB Data API to call a C++
 * shared library created using the MATLAB Compiler SDK.
 * Demonstrates passing matrices via the MATLAB Data API.
```

```

* Refer to the MATLAB Compiler SDK documentation for more
* information.
*
* Copyright 2017-Present The MathWorks, Inc.
*
*=====*/

// Include the header file required to use the generic
// interface for the C++ shared library generated by the
// MATLAB Compiler SDK.
#include "MatlabCppSharedLib.hpp"
#include <iostream>
#include <numeric> // for iota

namespace mc = matlab::cpplib;
namespace md = matlab::data;

std::u16string convertAsciiToUtf16(const std::string & asciiStr);

template <typename T>
void writeMatrix(std::ostream & ostr, const md::TypedArray<T> & matrix,
md::MemoryLayout layoutOfArray = md::MemoryLayout::ROW_MAJOR);

int mainFunc(std::shared_ptr<mc::MATLABApplication> app,
const int argc, const char * argv[]);

// The main routine. On the Mac, the main thread runs the system code, and
// user code must be processed by a secondary thread. On other platforms,
// the main thread runs both the system code and the user code.
int main(const int argc, const char * argv[])
{
    int ret = 0;
    try {
        auto mode = mc::MATLABApplicationMode::IN_PROCESS;
        const std::string STR_OPTIONS = "-nojvm";
        const std::u16string U16STR_OPTIONS = convertAsciiToUtf16(STR_OPTIONS);
        std::vector<std::u16string> options = {U16STR_OPTIONS};
        auto matlabApplication = mc::initMATLABApplication(mode, options);
        ret = mc::runMain(mainFunc, std::move(matlabApplication), argc, argv);
        // Calling reset() on matlabApplication allows the user to control
        // when it is destroyed, which automatically cleans up its resources.
        // Here, the object would go out of scope and be destroyed at the end
        // of the block anyway, even if reset() were not called.
        // Whether the matlabApplication object is explicitly or implicitly
        // destroyed, initMATLABApplication() cannot be called again within
        // the same process.
        matlabApplication.reset();
    } catch(const std::exception & exc) {
        std::cerr << exc.what() << std::endl;
        return -1;
    }
    return ret;
}

int mainFunc(std::shared_ptr<mc::MATLABApplication> app,
const int argc, const char * argv[])
{
    try {

```

```

// If using a compiler that supports the u"" prefix to indicate
// a char16_t *, you could simply pass u"libmatrix.ctf" as
// the second parameter to initMATLABLibrary(), and would
// not need to perform an extra step to convert from a
// narrow string. Visual C++ 2013 does not support the u""
// prefix, but later versions of Visual C++ do, as do other
// third-party compilers supported for use with MATLAB.
const std::string STR_CTF_NAME = "libmatrix.ctf";
const std::u16string U16STR_CTF_NAME = convertAsciiToUtf16(STR_CTF_NAME);

// The path to the CTF (library archive file) passed to
// initMATLABLibrary or initMATLABLibraryAsync may be either absolute
// or relative. If it is relative, the following will be prepended
// to it, in turn, in order to find the CTF:
// - the directory named by the environment variable
// CPPSHARED_BASE_CTF_PATH, if defined
// - the working directory
// - the directory where the executable is located
// - on Mac, the directory three levels above the directory
// where the executable is located

// If the CTF is not in one of these locations, do one of the following:
// - copy the CTF
// - move the CTF
// - change the working directory ("cd") to the location of the CTF
// - set the environment variable to the location of the CTF
// - edit the code to change the path
auto lib = mc::initMATLABLibrary(app, U16STR_CTF_NAME);
md::ArrayFactory factory;
const size_t NUM_ROWS = 3;
const size_t NUM_COLS = 3;
md::TypedArray<double> doubles = factory.createArray<double>({NUM_ROWS, NUM_COLS},
    {1.0, 2.0, 3.0,
     4.0, 5.0, 6.0,
     7.0, 8.0, 9.0});

// Note that the matrix is interpreted as being in column-major order
// (the MATLAB convention) rather than row-major order (the C++
// convention). Thus, the output from the next two lines of code will
// look like this:
//     The original matrix is:
//     1 4 7
//     2 5 8
//     3 6 9
// If you want to work with a matrix that looks like this:
//     1 2 3
//     4 5 6
//     7 8 9
// you can either store the data as follows:
//     md::TypedArray<double> doubles =
//     factory.createArray<double>({NUM_ROWS, NUM_COLS},
//     {1.0, 4.0, 7.0,
//      2.0, 5.0, 8.0,
//      3.0, 6.0, 9.0});
// or apply the MATLAB transpose function to the original matrix.
std::cout << "The original matrix is: " << std::endl;
writeMatrix<double>(std::cout, doubles);

```

```

std::vector<md::Array> matrices{doubles, doubles};
std::cout << "The sum of the matrix with itself is: " << std::endl;
auto sum = lib->feval("addmatrix", 1, matrices);
// The feval call returns a vector (of length 1) of md::Array objects.
writeMatrix<double>(std::cout, sum[0]);

std::cout << "The product of the matrix with itself is: " << std::endl;
auto product = lib->feval("multiplymatrix", 1, matrices);
writeMatrix<double>(std::cout, product[0]);

std::cout << "The eigenvalues of the original matrix are: " << std::endl;
std::vector<md::Array>single_matrix{doubles};
auto eigenvalues = lib->feval("eigmatrix", 1, single_matrix);
writeMatrix<double>(std::cout, eigenvalues[0]);

// This part of the code shows how createBuffer and createArrayFromBuffer
// can be used to convert from row-major to column-major order.
auto colMajorMatrixBuffer = factory.createBuffer<int>(6);
// The following call writes the values 100, 101, 102, 103, 104, 105
// into colMajorMatrixBuffer.
std::iota(colMajorMatrixBuffer.get(), colMajorMatrixBuffer.get() + 6, 100);
auto colMajorMatrixArray = factory.createArrayFromBuffer({2, 3},
    std::move(colMajorMatrixBuffer), md::MemoryLayout::COLUMN_MAJOR);
// OUTPUT:
// The original contents of the column-major matrix are:
//     100 102 104
//     101 103 105
std::cout << "The original contents of the column-major matrix are: " << std::endl;
writeMatrix<int>(std::cout, colMajorMatrixArray);
std::vector<md::Array> colMajorMatrixArrays{colMajorMatrixArray,
    colMajorMatrixArray};

// OUTPUT:
// The sum of the column-major matrix with itself is:
// 200 204 208
// 202 206 210
std::cout << "The sum of the column-major matrix with itself is: " << std::endl;
auto sumOfColMajorMatrixArrays = lib->feval("addmatrix", 1, colMajorMatrixArrays);
// The feval call returns a vector (of length 1) of md::Array objects.
writeMatrix<int>(std::cout, sumOfColMajorMatrixArrays[0]);

auto rowMajorMatrixBuffer = factory.createBuffer<int>(6);
std::iota(rowMajorMatrixBuffer.get(), rowMajorMatrixBuffer.get() + 6, 100);
auto rowMajorMatrixArray = factory.createArrayFromBuffer({3, 2},
    std::move(rowMajorMatrixBuffer), md::MemoryLayout::ROW_MAJOR);
// OUTPUT:
// The original contents of the row-major matrix are:
// 100 101
// 102 103
// 104 105
std::cout << "The original contents of the row-major matrix are: " << std::endl;
writeMatrix<int>(std::cout, rowMajorMatrixArray);
std::vector<md::Array> rowMajorMatrixArrays{rowMajorMatrixArray, rowMajorMatrixArray};

// OUTPUT:
// The sum of the row-major matrix with itself is:
// 200 202
// 204 206

```

```

        // 208 210
        std::cout << "The sum of the row-major matrix with itself is: " << std::endl;
        auto sumOfRowMajorMatrixArrays = lib->feval("addmatrix", 1, rowMajorMatrixArrays);
        // The feval call returns a vector (of length 1) of md::Array objects.
        writeMatrix<int>(std::cout, sumOfRowMajorMatrixArrays[0]);
    } catch(const std::exception & exc) {
        std::cerr << exc.what() << std::endl;
        return -1;
    }
    return 0;
}

std::u16string convertAsciiToUtf16(const std::string & asciiStr)
{
    return std::u16string(asciiStr.cbegin(), asciiStr.cend());
}

template <typename T>
void writeMatrix(std::ostream & ostr, const md::TypedArray<T> & matrix,
                md::MemoryLayout layoutOfArray /*= md::MemoryLayout::ROW_MAJOR*/)
{
    md::ArrayDimensions dims = matrix.getDimensions();
    if (dims.size() != 2)
    {
        std::ostringstream ostrstrm;
        ostrstrm << "Number of dimensions must be 2; actual number: " << dims.size();
        throw std::runtime_error(ostrstrm.str());
    }

    switch(layoutOfArray)
    {
        case md::MemoryLayout::ROW_MAJOR:
            for (size_t row = 0; row < dims[0]; ++row)
            {
                for (size_t col = 0; col < dims[1]; ++col)
                {
                    std::cout << matrix[row][col] << " ";
                }
                std::cout << std::endl;
            }
            break;

        case md::MemoryLayout::COLUMN_MAJOR:
            for (size_t col = 0; col < dims[1]; ++col)
            {
                for (size_t row = 0; row < dims[0]; ++row)
                {
                    std::cout << matrix[row][col] << " ";
                }
                std::cout << std::endl;
            }
            break;
    }
    std::cout << std::endl;
}

```

## **See Also**

### **More About**

- [“Generate a C++ MATLAB Data API Shared Library and Build a C++ Application”](#)





# Strongly Typed Cross Language Interface

---

## C++ MATLAB Data API Shared Library Support for Strongly Typed MATLAB Code

When creating C++ shared libraries from MATLAB functions or classes, you can stipulate how to represent MATLAB data types in C++ application code by using standard and custom data type mappings between MATLAB and C++. To specify the data type requirements, you use an `arguments` block within a MATLAB function or a `properties` block and `arguments` block within a MATLAB class. For example, if your C++ application code uses a `float` data type to represent a real scalar double value, its equivalent representation in MATLAB is `(1,1) single {mustBeReal}`.

### Sample MATLAB Function with Strongly Typed Data

```
function r = stronglyTypedFun(num)
arguments
    num (1,1) single {mustBeReal}
end
r = magic(num);
```

### Sample MATLAB Class with Strongly Typed Data

```
classdef MyPosition
    properties
        X (1,1) double {mustBeReal}
        Y (1,1) double {mustBeReal}
    end
end
```

For details, see “Data Type Mappings Between C++ and Strongly Typed MATLAB Code” on page 11-13.

When you compile a strongly typed MATLAB function, class, or package, MATLAB Compiler SDK generates a C++ shared library header (`.hpp` file) and a deployable archive (`.ctf` file). To generate the header file from the MATLAB command prompt, enter the `mcc` command using this syntax:

```
mcc -W 'cpplib:<library_name>,generic' <MATLAB file(s) and/or package folder(s)> -d <output folder>
```

---

**Tip** To generate the header file using the **Library Compiler** app:

- 1 In the **Type** section of the toolstrip, click **C++ Shared Library**.
  - 2 In the **Exported Functions** section of the toolstrip, add the relevant MATLAB files.
  - 3 In the **API selection** section, select the **Create interfaces that use the MATLAB Data API and Strongly Typed Interface** option, and click **Package**.
- 

The header file (`.hpp`) is generated in the same place as the deployable archive (`.ctf`) in the `v2\generic_interface` folder.

The generated header file:

- Maps strongly typed MATLAB data types to C++ data types. For an example, see “Create C++ MATLAB Data API Shared Library Header from Strongly Typed MATLAB Function” on page 11-5.

- Contains C++ namespaces that correspond to MATLAB package directories of the same name. For an example, see “Create C++ MATLAB Data API Shared Library Header from Strongly Typed MATLAB Classes Contained in Package” on page 11-8.
- Contains C++ classes that correspond to MATLAB classes of the same name.
- Contains public C++ methods that correspond to the public methods of MATLAB classes. The method names are unchanged and can be used as is in the C++ application code. These aligned names eliminate the need for intermediate layer top-level functions that call the class methods through an `feval` function execution.
- Contains C++ get and set methods for public properties of MATLAB classes. The property names of MATLAB classes are prepended with `get` or `set`. For example, if the property name in a MATLAB class is `UpperLeft`, the corresponding C++ method names are `getUpperLeft` and `setUpperLeft`.

### Mapping of Strongly Typed MATLAB Class to C++ Header File

Strongly Typed MATLAB Class	Snippet of C++ Header File
<pre> classdef MyRectangle      properties         UpperLeft (1,1) shapes.MyPosition         LowerRight (1,1) shapes.MyPosition     end     methods         function R = enlarge(R, n)             arguments                 R (1,1) shapes.MyRectangle                 n (1,1) double {mustBeReal}             end             % code         end         function R = show(R)             arguments                 R (1,1) shapes.MyRectangle             end             % code         end     end end end </pre>	<pre> namespace shapes {     class MyRectangle : public MATLABObject&lt;MATLABCon     public:         // constructors         MyRectangle() : MATLABObject() {}          // code          // properties         shapes::MyPosition getUpperLeft() {             // code         }         void setUpperLeft(shapes::MyPosition obj) {             // code         }         shapes::MyPosition getLowerRight() {             // code         }         void setLowerRight(shapes::MyPosition obj) {             // code         }          // methods         matlab::data::Array show() {             // code         }          matlab::data::Array enlarge(double n) {             // code         }     }; } </pre>

The generated header file (.hpp file) and the `MatlabCppSharedLib.hpp` header file are included in the C++ application code using `#include` directives. You can then compile and run the application.

### Sample C++ Application Code Snippet

```
#include "MatlabCppSharedLib.hpp"
#include "output/cpp/v2/generic_interface/libshapesv2.hpp" //header file generated by mcc

int main(const int argc, char *argv[]) {
    try {
        // common starter code that can apply to any application
        auto mode = matlab::cpplib::MATLABApplicationMode::IN_PROCESS;
        std::vector<std::u16string> OPTIONS = {"-nojvm"};
        auto appPtr = matlab::cpplib::initMATLABApplication(mode, OPTIONS);
        std::string ctfName(argv[1]);
        auto libPtr = matlab::cpplib::initMATLABLibrary(appPtr, std::u16string(ctfName.cbegin(),
        std::shared_ptr<MATLABControllerType> matlabPtr(std::move(libPtr));

        // application specific code that relies on the generated header
        shapes::MyPosition p1(matlabPtr);
        ...
        shapes::MyRectangle r1(matlabPtr);
        ...
    }
}
```

---

**Tip**

- When writing C++ application code, you must include the header file (.hpp file) generated by the `mcc` command or the **Library Compiler** app and the `MatlabCppSharedLib.hpp` header file using `#include` directives.
- Your MATLAB code must be strongly typed to leverage all the features of the interface. Otherwise, data mapping between MATLAB and C++ is absent, and you cannot use native C++ data types.
- During the compilation process, strongly typed information is retrieved only from `arguments` and `properties` blocks. The information retrieved consists of array size, type, and whether it is a real number.

---

**See Also**

`arguments` | `properties`

**More About**

- “Create C++ MATLAB Data API Shared Library Header from Strongly Typed MATLAB Function” on page 11-5
- “Create C++ MATLAB Data API Shared Library Header from Strongly Typed MATLAB Classes Contained in Package” on page 11-8
- “Data Type Mappings Between C++ and Strongly Typed MATLAB Code” on page 11-13
- “Function Argument Validation”
- “Argument Validation Functions”
- “Validate Property Values”
- “Property Validation Functions”

## Create C++ MATLAB Data API Shared Library Header from Strongly Typed MATLAB Function

This example shows how to create a C++ MATLAB Data API shared library header from a strongly typed MATLAB function and integrate it with sample C++ application code. The target system does not require a licensed copy of MATLAB to run the application.

### Prerequisites

- Start this example by creating a new work folder that is visible to the MATLAB search path.
- Verify that you have a C++ compiler installed by typing `mbuild -setup` at the MATLAB command prompt.
- End users must have an installation of MATLAB Runtime to run the application. For details, see “Install and Configure MATLAB Runtime”.

For testing purposes, you can use an installation of MATLAB instead of MATLAB Runtime.

### Create Function in MATLAB

Create a MATLAB file named `stronglyTypedFactorial.m` with the following code:

```
function fact = stronglyTypedFactorial(n)
arguments
    n (1,1) uint64 {mustBeReal, mustBeLessThan(n,22)}
end
fact = 1;
for i = 1:n
    fact = fact*i;
end
end
```

Test the function at the MATLAB command prompt.

```
stronglyTypedFactorial(5)
```

```
ans =
    uint64
     120
```

### Generate C++ Shared Library Header Using the `mcc` Command

Generate the C++ shared library header using the `mcc` command. At the command prompt, type:

```
if ~exist('output/cpp', 'dir')
    mkdir output/cpp
end
mcc -W 'cpplib:stronglyTypedFactorial,generic' stronglyTypedFactorial.m -d output/cpp
```

The following files are created in the `v2 > generic_interface` folder:

- `readme.txt`
- `stronglyTypedFactorial.ctf`
- `stronglyTypedFactorialv2.hpp`

The `stronglyTypedFactorialv2.hpp` header file contains a C++ data array that accepts an argument of type `uint64_t`.

```
matlab::data::Array stronglyTypedFactorial(std::shared_ptr<MATLABControllerType> _matlabPtr, uint64_t n)
```

This `uint64_t` mapping matches the strongly typed `uint64` data type of the MATLAB argument.

```
n (1,1) uint64 {mustBeReal, mustBeLessThan(n,22)}
```

For details, see “Data Type Mappings Between C++ and Strongly Typed MATLAB Code” on page 11-13.

### **stronglyTypedFactorialv2.hpp**

```
#include "MatlabTypesInterface.hpp"
```

```
matlab::data::Array stronglyTypedFactorial(std::shared_ptr<MATLABControllerType> _matlabPtr, uint64_t n)
{
    matlab::data::ArrayFactory _arrayFactory;
    matlab::data::ArrayDimensions _dims1 = {1, 1};
    std::vector<matlab::data::Array> _args = {
        _arrayFactory.createArray<uint64_t>(_dims1, {n}) };
    matlab::data::Array _result = _matlabPtr->feval("stronglyTypedFactorial", _args);
    return _result;
}
```

## **Integrate C++ MATLAB Data API Shared Library Header with C++ Application**

Create a C++ application code file named `factApp.cpp` with the following code.

### **factApp.cpp**

```
// Include header files
#include "MatlabCppSharedLib.hpp"
#include "output/cpp/v2/generic_interface/stronglyTypedFactorialv2.hpp"
#include <iostream>

int main(const int argc, char *argv[])
{
    if (argc != 2) {
        std::cout << "Usage: " << argv[0] << " <path to .ctf file.>\n";
        return 0;
    }

    try {
        // Initialize MATLAB application
        auto mode = matlab::cpplib::MATLABApplicationMode::IN_PROCESS;
        std::vector<std::u16string> OPTIONS = {u"-nojvm"};
        auto appPtr = matlab::cpplib::initMATLABApplication(mode, OPTIONS);

        // Initialize MATLAB Runtime from the .ctf file
        std::string ctfName(argv[1]);
        auto libPtr = matlab::cpplib::initMATLABLibrary(appPtr, std::u16string(ctfName.cbegin(), ctfName.cend()));
        std::shared_ptr<MATLABControllerType> matlabPtr(std::move(libPtr));

        // Call the stronglyTypedFactorial MATLAB function
        matlab::data::TypedArray<uint64_t> output = stronglyTypedFactorial(matlabPtr, 5);
    }
}
```

```

        std::cout << "Factorial of 5 is " << output[0] << "\n";
        appPtr.reset();
        return 0;
    } catch (const std::exception & exc) {
        std::cout << exc.what() << std::endl;
        return 1;
    }
}

```

---

**Note** When writing C++ application code, you must include the header file (.hpp file) generated by the `mcc` command or the **Library Compiler** app and the `MatlabCppSharedLib.hpp` header file using `#include` directives.

---

Compile and link the C++ application at the MATLAB command prompt.

```
mbuild factApp.cpp -outdir output/cpp
```

Run the application from the system command prompt by passing the deployable archive (.ctf file) as an input. For testing purposes, you can run the application from the MATLAB command prompt.

```
!output\cpp\factApp.exe output\cpp\v2\generic_interface\stronglyTypedFactorial.ctf
```

```
Factorial of 5 is 120
```

## See Also

[arguments](#) | [properties](#)

## More About

- “C++ MATLAB Data API Shared Library Support for Strongly Typed MATLAB Code” on page 11-2
- “Create C++ MATLAB Data API Shared Library Header from Strongly Typed MATLAB Classes Contained in Package” on page 11-8
- “Data Type Mappings Between C++ and Strongly Typed MATLAB Code” on page 11-13
- “Function Argument Validation”
- “Argument Validation Functions”
- “Validate Property Values”
- “Property Validation Functions”

## Create C++ MATLAB Data API Shared Library Header from Strongly Typed MATLAB Classes Contained in Package

This example shows how to create a C++ shared library header from strongly typed MATLAB classes contained within a package and integrate it with sample C++ application code.

### Prerequisites

- Start this example by creating a new work folder that is visible to the MATLAB search path.
- Verify that you have a C++ compiler installed by typing `mbuild -setup` at the MATLAB command prompt.
- End users must have an installation of MATLAB Runtime to run the application. For details, see “Install and Configure MATLAB Runtime”.

For testing purposes, you can use an installation of MATLAB instead of MATLAB Runtime.

### Files

#### Location of Example Files

Example Files	<code>matlabroot\extern\examples\compilersdk\c_cpp\strongly_typed</code>
---------------	--

#### Purpose of Each Example File

Files	Purpose
<code>+shapes</code>	Package containing two classes: <code>MyPosition.m</code> and <code>MyRectangle.m</code>
<code>MyPosition.m</code>	A class within the <code>+shapes</code> package that accepts the X and Y coordinates of a point and creates a <code>MyPosition</code> object
<code>MyRectangle.m</code>	A class within the <code>+shapes</code> package that accepts two points specified as <code>MyPosition</code> objects and creates a <code>MyRectangle</code> object
<code>calculatearea.m</code>	A function that accepts a <code>MyRectangle</code> object as input and calculates the area of the rectangle
<code>shapes_mda.cpp</code>	C++ application code that integrates the header file generated by compiling the MATLAB code

Copy the example files to the current work folder.

```
appDir = fullfile(matlabroot, 'extern', 'examples', 'compilersdk', 'c_cpp', 'strongly_typed');
copyfile(appDir)
```

### Create Classes and Functions in MATLAB

- 1 Examine the code for `MyPosition.m`, `MyRectangle.m`, and `calculatearea.m`.
  - The `+shapes` package contains two MATLAB classes: `MyPosition.m` and `MyRectangle.m`.
  - The `calculatearea.m` MATLAB function located outside of the `+shapes` package accepts a `MyRectangle` object as input and calculates the area of the rectangle.



- 2 Create a MATLAB script named `runshapes.m` with the following code and execute it at the MATLAB command prompt. This script illustrates how the classes and function interact to generate an output.

### `runshapes.m`

```
% Import shapes package
import shapes.*;

%% Create Rectangle 1
% Create MyPosition object for point 1
p1 = MyPosition;
p1.X = 10;
p1.Y = 5;
% Create MyPosition object for point 2
p2 = MyPosition;
p2.X = 50;
p2.Y = 20;
% Create MyRectangle object
r1 = MyRectangle;
r1.UpperLeft = p1;
r1.LowerRight = p2;
% Calculate area of rectangle 1
a1 = calculatearea(r1);
% Display rectangle 1 info
disp('Rectangle 1')
disp(['Point 1 = ' '( ' num2str(p1.X,'%0.6f'), ', ' num2str(p1.Y,'%0.6f') ')'])
disp(['Point 2 = ' '( ' num2str(p2.X,'%0.6f'), ', ' num2str(p2.Y,'%0.6f') ')'])
disp(['Rectangle ' '( ' num2str(p1.X,'%0.6f'), ', ' num2str(p1.Y,'%0.6f') ')',...
      ' -> ' '( ' num2str(p2.X,'%0.6f'), ', ' num2str(p2.Y,'%0.6f') ')'])

%% Create Rectangle 2
% Create rectangle 2 by enlarging rectangle 1
r2 = r1.enlarge(10);
% Get positions of rectnagle 2
q1 = r2.UpperLeft();
q2 = r2.LowerRight();
% Calculate area of rectangle 2
a2 = calculatearea(r2);
% Display rectangle 2 info
disp('Rectangle 2')
disp(['Point 1 = ' '( ' num2str(q1.X,'%0.6f'), ', ' num2str(q1.Y,'%0.6f') ')'])
disp(['Point 2 = ' '( ' num2str(q2.X,'%0.6f'), ', ' num2str(q2.Y,'%0.6f') ')'])
disp(['Rectangle ' '( ' num2str(q1.X,'%0.6f'), ', ' num2str(q1.Y,'%0.6f') ')',...
      ' -> ' '( ' num2str(q2.X,'%0.6f'), ', ' num2str(q2.Y,'%0.6f') ')'])

%% Display the area of the two rectangles
disp(['Area of rectangle r1 = ' num2str(a1)])
disp(['Area of rectangle r2 = ' num2str(a2)])

runshapes

Rectangle 1
Point 1 = (10.000000,5.000000)
Point 2 = (50.000000,20.000000)
Rectangle (10.000000,5.000000) -> (50.000000,20.000000)
Rectangle 2
Point 1 = (0.000000,-5.000000)
```

```
Point 2 = (60.000000,30.000000)
Rectangle (0.000000,-5.000000) -> (60.000000,30.000000)
Area of rectangle r1 = 600
Area of rectangle r2 = 2100
```

## Generate C++ Shared Library Header Using the mcc Command

```
if ~exist('output/cpp','dir')
    mkdir output/cpp
end
mcc -W 'cpplib:libshapes,generic' +shapes/MyPosition.m +shapes/MyRectangle.m calculatearea.m -d
```

The following files are created in the output>cpp>v2>generic\_interface folder:

- readme.txt
- libshapes.ctf
- libshapesv2.hpp

For details, see “Data Type Mappings Between C++ and Strongly Typed MATLAB Code” on page 11-13.

### libshapesv2.hpp

```
#include "MatlabTypesInterface.hpp"

namespace shapes {
    class MyPosition : public MATLABObject<MATLABControllerType> {
    public:

        // constructors
        MyPosition() : MATLABObject() {}

        MyPosition(std::shared_ptr<MATLABControllerType> matlabPtr) :
            MATLABObject(matlabPtr, u"shapes.MyPosition")
        {}

        MyPosition(std::shared_ptr<MATLABControllerType> matlabPtr, matlab::data::Array obj) :
            MATLABObject(matlabPtr, obj)
        {}

        // properties
        double getX() { return MATLABGetScalarProperty<double>(u"X"); }
        void setX(double value) { return MATLABSetScalarProperty(u"X", value); }
        double getY() { return MATLABGetScalarProperty<double>(u"Y"); }
        void setY(double value) { return MATLABSetScalarProperty(u"Y", value); }

        // methods
    };
}

namespace shapes {
    class MyRectangle : public MATLABObject<MATLABControllerType> {
    public:

        // constructors
        MyRectangle() : MATLABObject() {}
    };
}
```

```

MyRectangle(std::shared_ptr<MATLABControllerType> matlabPtr) :
    MATLABObject(matlabPtr, u"shapes.MyRectangle")
{}

MyRectangle(std::shared_ptr<MATLABControllerType> matlabPtr, matlab::data::Array obj) :
    MATLABObject(matlabPtr, obj)
{}

// properties
shapes::MyPosition getUpperLeft() {
    matlab::data::Array obj = MATLABGetScalarProperty<matlab::data::Array>(u"UpperLeft");
    return shapes::MyPosition(m_matlabPtr, obj);
}
void setUpperLeft(shapes::MyPosition obj) {
    MATLABSetScalarProperty(u"UpperLeft", matlab::data::Array(obj));
}
shapes::MyPosition getLowerRight() {
    matlab::data::Array obj = MATLABGetScalarProperty<matlab::data::Array>(u"LowerRight");
    return shapes::MyPosition(m_matlabPtr, obj);
}
void setLowerRight(shapes::MyPosition obj) {
    MATLABSetScalarProperty(u"LowerRight", matlab::data::Array(obj));
}

// methods
matlab::data::Array show() {
    matlab::data::ArrayFactory _arrayFactory;
    std::vector<matlab::data::Array> _args = {
        m_object };
    matlab::data::Array _result = MATLABCallOneOutputMethod(u"show", _args);
    return _result;
}

matlab::data::Array enlarge(double n) {
    matlab::data::ArrayFactory _arrayFactory;
    matlab::data::ArrayDimensions _dims1 = {1, 1};
    std::vector<matlab::data::Array> _args = {
        m_object,
        _arrayFactory.createArray<double>(_dims1, {n}) };
    matlab::data::Array _result = MATLABCallOneOutputMethod(u"enlarge", _args);
    return _result;
}

};
}

matlab::data::Array calculatearea(std::shared_ptr<MATLABControllerType> _matlabPtr, shapes::MyRe
    matlab::data::ArrayFactory _arrayFactory;
    matlab::data::ArrayDimensions _dims1 = {1, 1};
    std::vector<matlab::data::Array> _args = {
        rect };
    matlab::data::Array _result = _matlabPtr->feval(u"calculatearea", _args);
    return _result;
}

```

## Integrate C++ MATLAB Data API Shared Library Header with C++ Application

- 1 Examine the C++ application code contained in the `shapes_mda.cpp` file.

---

**Note** When writing C++ application code, you must include the header file (`.hpp` file) generated by the `mcc` command or the **Library Compiler** app and the `MatlabCppSharedLib.hpp` header file using `#include` directives.

---

- 2 Compile and link the C++ application at the MATLAB command prompt.

```
mbuild shapes_mda.cpp -outdir output/cpp
```

- 3 Run the application from the system command prompt by passing the deployable archive (`.ctf` file) as an input. Before running the application at the system command prompt, verify that you have MATLAB Runtime installed on your machine prior to running the application at the system command prompt. For details, see “Install and Configure MATLAB Runtime”.

For testing purposes, you can run the application from the MATLAB command prompt.

```
!output\cpp\shapes_mda.exe output\cpp\v2\generic_interface\libshapes.ctf
```

```
Rectangle 1
Point (10.000000, 5.000000)
Point (50.000000, 20.000000)
Rectangle (10.000000, 5.000000) -> (50.000000, 20.000000)
Rectangle 2
Point (0.000000, -5.000000)
Point (60.000000, 30.000000)
Rectangle (0.000000, -5.000000) -> (60.000000, 30.000000)
Area of rectangle r1 = 600
Area of rectangle r2 = 2100
```

### See Also

[arguments](#) | [properties](#)

### More About

- “C++ MATLAB Data API Shared Library Support for Strongly Typed MATLAB Code” on page 11-2
- “Create C++ MATLAB Data API Shared Library Header from Strongly Typed MATLAB Function” on page 11-5
- “Data Type Mappings Between C++ and Strongly Typed MATLAB Code” on page 11-13
- “Function Argument Validation”
- “Argument Validation Functions”

## Data Type Mappings Between C++ and Strongly Typed MATLAB Code

### C++ to MATLAB

This table shows the mapping of C++ data types to strongly typed MATLAB data types.

Description	C++ Data Type	MATLAB Data Type Representation
Real double scalar	double	(1, 1) double {mustBeReal}
Real double vector	std::vector<double>	(1, :) double {mustBeReal}
Complex double scalar Can be passed to MATLAB as plain real through C++ overloading	std::complex<double>	(1, 1) double
Complex double vector Can be passed to MATLAB as plain real through C++ overloading	std::vector<std::complex<double>>	(1, :) double
Real single scalar	float	(1, 1) single {mustBeReal}
Real single vector	std::vector<float>	(1, :) single {mustBeReal}
Complex single scalar Can be passed to MATLAB as plain real through C++ overloading	std::complex<float>	(1, 1) single
Complex single vector Can be passed to MATLAB as plain real through C++ overloading	std::vector<std::complex<float>>	(1, :) single

Description	C++ Data Type	MATLAB Data Type Representation
Real integer scalars	int8_t int16_t int32_t int64_t uint8_t uint16_t uint32_t uint64_t	(1, 1) int8 {mustBeReal} (1, 1) int16 {mustBeReal} (1, 1) int32 {mustBeReal} (1, 1) int64 {mustBeReal} (1, 1) uint8 {mustBeReal} (1, 1) uint16 {mustBeReal} (1, 1) uint32 {mustBeReal} (1, 1) uint64 {mustBeReal}
Real integer vectors	std::vector<int8_t> std::vector<int16_t> std::vector<int32_t> std::vector<int64_t> std::vector<uint8_t> std::vector<uint16_t> std::vector<uint32_t> std::vector<uint64_t>	(1, :) int8 {mustBeReal} (1, :) int16 {mustBeReal} (1, :) int32 {mustBeReal} (1, :) int64 {mustBeReal} (1, :) uint8 {mustBeReal} (1, :) uint16 {mustBeReal} (1, :) uint32 {mustBeReal} (1, :) uint64 {mustBeReal}

Description	C++ Data Type	MATLAB Data Type Representation
Complex integer scalars	<code>std::complex&lt;int8_t&gt;</code>	(1, 1) int8
	<code>std::complex&lt;int16_t&gt;</code>	(1, 1) int16
	<code>std::complex&lt;int32_t&gt;</code>	(1, 1) int32
	<code>std::complex&lt;int64_t&gt;</code>	(1, 1) int64
	<code>std::complex&lt;uint8_t&gt;</code>	(1, 1) uint8
	<code>std::complex&lt;uint16_t&gt;</code>	(1, 1) uint16
	<code>std::complex&lt;uint32_t&gt;</code>	(1, 1) uint32
	<code>std::complex&lt;uint64_t&gt;</code>	(1, 1) uint64
Complex integer vectors	<code>std::vector&lt;std::complex&lt;int8_t&gt;&gt;</code>	(1, :) int8 (1, :) int16
	<code>std::vector&lt;std::complex&lt;int16_t&gt;&gt;</code>	(1, :) int32
	<code>std::vector&lt;std::complex&lt;int32_t&gt;&gt;</code>	(1, :) int64 (1, :) uint8
	<code>std::vector&lt;std::complex&lt;int64_t&gt;&gt;</code>	(1, :) uint16
	<code>std::vector&lt;std::complex&lt;uint8_t&gt;&gt;</code>	(1, :) uint32 (1, :) uint64
	<code>std::vector&lt;std::complex&lt;uint16_t&gt;&gt;</code>	
	<code>std::vector&lt;std::complex&lt;uint32_t&gt;&gt;</code>	
	<code>std::vector&lt;std::complex&lt;uint64_t&gt;&gt;</code>	
Logical scalar	<code>bool</code>	(1, 1) logical
Logical vector	<code>std::vector&lt;bool&gt;</code>	(1, :) logical
Char scalar	<code>char16_t</code>	(1, 1) char
Char vector	<code>std::u16string</code>	(1, :) char
String	<code>std::u16string</code>	(1, 1) string
String vector	<code>std::vector&lt;std::u16string&gt;</code>	(1, :) string
Enum scalar	C++ scoped enumeration	(1, 1) MyEnumClass
Enum vector	<code>std::vector</code> of scoped enumeration	(1, :) MyEnumClass

## MATLAB to C++

This table shows the mapping of strongly typed MATLAB data types to C++ data types.

Description	MATLAB Data Type	C++ Data Type Representation
Real double scalar	(1, 1) double {mustBeReal}	double or vector<double>
Real double vector	(1, :) double {mustBeReal}	std::vector<double>
Complex double scalar Stored as a complex number, even if result is real	(1, 1) double	std::complex<double>
Complex double vector Stored in MATLAB Data Array if result is real	(1, :) double	matlab::data::Array  If result contains DOUBLE or COMPLEX_DOUBLE:  matlab::data::ArrayType
Real single scalar	(1, 1) single {mustBeReal}	float
Real single vector	(1, :) single {mustBeReal}	std::vector<float>
Complex single scalar	(1, 1) single	std::complex<float>
Complex single vector	(1, :) single	matlab::data::Array  If result contains SINGLE or COMPLEX_SINGLE:  matlab::data::ArrayType



Description	MATLAB Data Type	C++ Data Type Representation
Real integer scalars	(1, 1) int8 {mustBeReal} (1, 1) int16 {mustBeReal} (1, 1) int32 {mustBeReal} (1, 1) int64 {mustBeReal} (1, 1) uint8 {mustBeReal} (1, 1) uint16 {mustBeReal} (1, 1) uint32 {mustBeReal} (1, 1) uint64 {mustBeReal}	int8_t int16_t int32_t int64_t uint8_t uint16_t uint32_t uint64_t
Real integer vectors	(1, :) int8 {mustBeReal} (1, :) int16 {mustBeReal} (1, :) int32 {mustBeReal} (1, :) int64 {mustBeReal} (1, :) uint8 {mustBeReal} (1, :) uint16 {mustBeReal} (1, :) uint32 {mustBeReal} (1, :) uint64 {mustBeReal}	std::vector<int8_t> std::vector<int16_t> std::vector<int32_t> std::vector<int64_t> std::vector<uint8_t> std::vector<uint16_t> std::vector<uint32_t> std::vector<uint64_t>

<b>Description</b>	<b>MATLAB Data Type</b>	<b>C++ Data Type Representation</b>
Complex integer scalars	(1, 1) int8	std::complex<int8_t>
	(1, 1) int16	std::complex<int16_t>
	(1, 1) int32	std::complex<int32_t>
	(1, 1) int64	std::complex<int64_t>
	(1, 1) uint8	std::complex<uint8_t>
	(1, 1) uint16	std::complex<uint16_t>
	(1, 1) uint32	std::complex<uint32_t>
	(1, 1) uint64	std::complex<uint64_t>
Complex integer vectors	(1, :) int8	matlab::data::Array
	(1, :) int16	If result contains INT8, COMPLEX_INT8, INT16, COMPLEX_INT16, INT32, COMPLEX_INT32, INT64, COMPLEX_INT64, UINT8, COMPLEX_UINT8, UINT16, COMPLEX_UINT16, UINT32, COMPLEX_UINT32, UINT64, or COMPLEX_UINT64:  matlab::data::ArrayType
	(1, :) int32	
	(1, :) int64	
	(1, :) uint8	
	(1, :) uint16	
	(1, :) uint32	
	(1, :) uint64	
Logical scalar	(1, 1) logical	
Logical vector	(1, :) logical	std::vector<bool>
Char scalar	(1, 1) char	char16_t
Char vector	(1, :) char	std::u16string
String	(1, 1) string	std::u16string
String vector	(1, :) string	std::vector<std::u16string>
Enum scalar	(1, 1) MyEnumClass	scoped enumeration
Enum vector	(1, :) MyEnumClass	std::vector of scoped enumeration
MATLAB cell scalar	(1, 1) cell	matlab::data::Array containing CELL matlab::data::ArrayType
MATLAB cell vector	(1, :) cell	matlab::data::Array containing CELL matlab::data::ArrayType

Description	MATLAB Data Type	C++ Data Type Representation
MATLAB struct	(1, 1) struct	matlab::data::Array containing STRUCT matlab::data::ArrayType
MATLAB struct	(1, :) struct	matlab::data::Array containing STRUCT matlab::data::ArrayType

## See Also

arguments | properties

## More About

- “Function Argument Validation”
- “Argument Validation Functions”

