Masarykova univerzita
Fakulta informatiky

Memory Management in MATLAB and External C++ Modules

Master's thesis

Josef Pacula

Brno, 2012

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Josef Pacula

Advisor: RNDr. Petr Matula, Ph.D.

Acknowledgement

First and foremost I would like to express enormous gratefulness to my thesis advisor RNDr. Petr Matula, Ph.D. for his outstanding patience, countless advice, and excellent guidance. One simply could not wish for a better or friendlier supervisor.

Secondly, I thank my colleague and friend Honza Urban for the flawless team work, great cooperation, and useful consultation of many topics.

Next, I thank Jan Houska for useful MATLAB product support and advice and Joel Andersson for insight and experiences.

Credit goes also to my work colleagues, namely Boris Parák, Miroslav Sliacky, and Filip Hubík for their indulgence during my elaboration on this thesis. Go MetaCloud Team. I would like to thank Petra for enduring the process and being with me.

Last, but not least, I am grateful to my parents, Marie and Josef, and my sister Hana. They have always supported me during my whole life. Mom, dad, sister, thank you.

Abstract

This master's thesis concerns memory management in MATLAB external C++ modules. It describes how MATLAB manages memory generally. It discusses possible methods of sharing memory between different interfaced C++ modules. This allows for interfacing of certain C++ features, such as classes and global variables to MATLAB. The text describes in greater detail the most suitable method which we implemented in the MATLAB module for SWIG (Simple Wrapper Interface Generator).

Keywords

MATLAB memory management, memory persistence, SWIG language module, C/C++ interface, C++ name mangling, automated wrapper code generation

Contents

1 Introduction

MATLAB is a commercial software very often used in scientific and technical computing. There are scientific libraries written in C/C++ which are not accessible from MATLAB without a properly written interface layer. The connection between different software packages is often realized by a layer called Application Programming Interface (API). API is a set of functions provided by a software component, which can be used for communication between different components. The interfacing code can be written manually by a programmer. However, there are tools that can automate wrapper code generation and significantly decrease the necessary manual work, for example SWIG (Simplified Wrapper Interface Generator) [14].

SWIG is a software development tool that connects programs written in C/C++ with a variety of high level programming languages, for example Java or Python. For each supported language a wrapper code can be generated based on special files describing library interface. However, MATLAB is not among target languages.

My Bachelor thesis [35] and Jan Urban's Bachelor thesis [40] concern work that we did to create a SWIG MATLAB module, which can automatically generate an interface to a specific library (Morph-M) via SWIG. The module implemented in our bachelor theses was based on MatlabConversion class developed by Andor Pathó [36], and this class must be modified to correctly map complex data types. Therefore the previous module cannot be used for a general C/C++ library without modification.

Development of a general SWIG MATLAB module, which would not be linked to a specific C/C++ library is a challenging task. We with Jan have redesigned and rewritten our previous MATLAB module to support general C/C++ libraries. While developing the general MATLAB module two main topics emerged: memory management and mapping of different data types between C++ and MATLAB. These topics are described in our master theses.

This thesis describes details of MATLAB memory management, memory persistence, and memory related issues of MATLAB external interfaces. It explores possible strategies of wrapping classes and other C++ language features, discusses advantages and disadvantages of the memory management strategies and describes the strategy that we selected in details. All topics are discussed with respect to automatic generation of the wrapper code.

Jan's thesis [41] focuses on the mapping of language features between

MATLAB and C++.

This thesis is organized into eight chapters. The second gives a brief introduction to memory management in general. The third lays out specifics of MATLAB memory management and its type system. The fourth chapter gives a summary of external MATLAB interfaces relevant for this thesis. The fifth describes necessary terminology for the next chapters, SWIG work flow overview, and SWIG basic principles. Design principles and a list of viable approaches to memory persistence and wrapper external interface follow in the sixth and seventh chapters, respectively.

2 Memory Management

Modern operating systems introduce concept of virtual memory of a process, which allows separation of logical memory as perceived by the user from physical memory. This dual memory view relaxes the necessity of memory free space and fragmentation management for the user. Virtual memory appears to be contiguous to the user. Memory management unit maps the logical pages to physical page frames in memory [38].

The virtual address space of a process (Figure 2.1) is a logical layout of a process in the memory and consists of 5 regions:

Text segment (also **text**, **code**) containing executable instructions of the process

Data segment of fixed size determined on compile time, storing global and static variables

Heap segment for dynamic memory allocations (expanding as needed)

Stack (also **call stack**) storing local variables declared in a block or function, behaving in *last in, first out*[1] manner

Sparse memory is region to which shared libraries, files, and shared memory pages are mapped

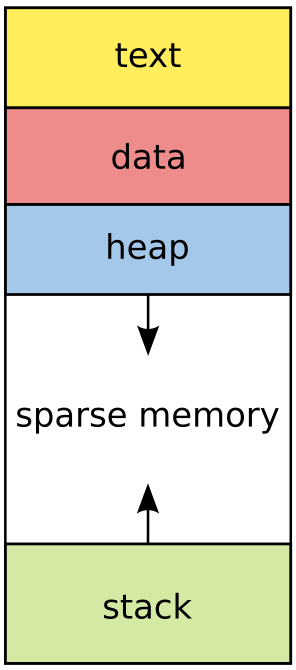


Figure 2.1: Virtual memory layout.

2.1 Manual Memory Management

Almost all modern programming languages make use of dynamic memory allocation. This allows data to be allocated and deallocated even if the total size of the data was unknown at compile time, or if the lifetime of allocated data exceeds the lifetime of the subroutine which allocated the data [32].

Any long running program needs to recover the allocated memory. The basic strategy is to use manual deallocation, where the used memory is

1. Last in, first out is a paraphrase of stack data structure axioms [34]. The latest inserted element is removed first.

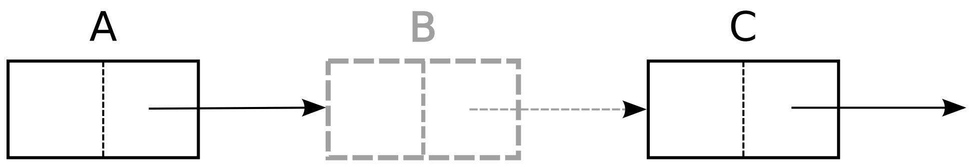


Figure 2.2: Deallocation of B in a linked list causes that A is holding a dangling pointer and C (and all its successors) are no longer referenced from anywhere and result in memory leak.

reclaimed by using explicit deallocation (for example command `free` in C or `delete` in C++). Manual memory deallocation is prone to programmer errors, which may present themselves in two ways: dangling pointers and memory leaks.

Dangling pointers arise in situations where a resource (an object or a block of memory) that is referenced from outside is deallocated. Then the outside reference points to no longer valid memory address. Figure 2.2 illustrates this phenomena. The figure also displays how memory leak occurs while the program loses a reference to the memory; when B is deallocated, C and all successors are no longer referenced and cannot be accessed. As illustrated in the Figure, single deallocation may lead to both dangling pointers and memory leaks.

Errors with memory management are more prevalent when multiple references to data are held. Safe deallocation is not just about taking more care. Deallocation of heap allocated memory is complex because as pointed out in [42] "liveness is a global property", while decision on calling `delete` or `free` is a local one.

There were several attempts to address this challenge in C++. Special pointers named *auto pointer* were one of the options [16]. However structures like that have limitations. These pointers could not be used with Standard Template Library (STL) and have been deprecated in the latest C++ ISO standard [31].

George Belotsky takes a different approach and appeals to strict programming habits rather than technology [19]. Belotsky puts stress on correct management of object ownership, and returning objects by value (copying the full contents of memory/object rather than returning a reference). Another principle is avoiding heap allocation altogether and preferring stack allocation. This way, on stack unwinding[2] the memory is properly reclaimed.

2. Stack unwinding refers to the action of discarding a top memory frame from the call

While good programming habits may limit/eliminate memory management errors, they also do so at the cost of increased overhead and loss of memory sharing. The latest C++ ISO standard updates and adds smart pointer capabilities [31]. The new reference counted `shared_ptr` pointer cannot be used in self-referencing data structures (such as graphs) [32].

With such a range of techniques and technologies, not only the collaborators of a library, but also third party components used in a project have to agree on a common approach.

2.2 Garbage Collection

Many of memory management issues described above are resolved by garbage collection, which is one of the forms of automatic dynamic memory management. Garbage collection assumes responsibility for dynamic memory reclamation, which eliminates dangling pointers and memory leaks. Moreover, garbage collection translates programmer's focus away from technicalities to the algorithmic problem in question, hence reduce development costs [20]. Common disadvantages of garbage collection usually include:

- Consumption of computing resources
- Unpredictable localized peaks in processor and memory utilization (unacceptable in applications such as real-time systems or user interactive systems)

The first entry is disputable, as for typical heap sizes garbage collection overhead increases by 17% on average compared to manual deallocation [30]. The article concerns benchmarking Java garbage collection algorithms and states that for large enough heap segment (five times the minimum amount), the execution time of garbage collected program matches the time of program with manual deallocation.

One example algorithm of garbage collection is reference counting. This algorithm maintains a simple invariant: an object needs to be allocated in memory as long as the number of references to that object is greater than zero. In reference counting the garbage collector keeps an integer with a reference count for each object. If a reference to an object is copied, the reference count is incremented, if a reference is deleted, the count is decremented.

stack of a process on function return [28].

A slight modification of this algorithm is reference listing, which holds a set of clients believed to be holding a reference to an object in question. If a new reference to an object is created, the holder of the reference is added to the reference set of the object. Deleting a reference removes the reference holder from the set of the referred object. As soon as the reference set is empty, the object can be deallocated. This improves the reference counting algorithm, because operations of set insertion and deletion are idempotent[3] whereas counter arithmetic is not [32].

2.3 Copy on Write

Data copying can be resource consuming. Copy on Write (COW) mechanism is data copying optimization strategy in programming. The central idea is to postpone the actual data duplication to as late as possible, *i.e.*, to the moment where one of the copy holders modifies the data, thereby reduce the time needed to make a copy. An example of COW application is new process startup in modern operating systems including Linux, Microsoft Windows, and Solaris [38, 27]:

1. To create a new process, parent process is forked.
2. The virtual memory of the new process is assigned physical memory pages of the parent process.
3. The assigned memory is marked "Copy on Write".
4. Both processes share the physical memory pages for reading as long as the memory is not modified by either process.
5. An attempt of "Copy on Write" memory page modification by either process is interrupted by the kernel and the page is first duplicated, whereat the modification proceeds.

3. An unary operation $\otimes$ is idempotent if and only if $\otimes \circ \otimes = \otimes$.

3 MATLAB Memory Management

This chapter contains necessary information to understand MATLAB memory management. In particular, topics related to the life cycle of MATLAB variables, the management of its external modules, and memory allocation and deallocation details. The chapter is partly based on MATLAB documentation [12] and partly on numerous experiments which we performed to understand all details needed to design and implement our general SWIG MATLAB module.

3.1 Variables and Workspaces

All variables in MATLAB, be it scalars, vectors, matrices, strings, cell arrays, structures, objects, are internally represented by a single object type, called $mxArray$. This data type gathers information such as (but not only):

- Object type
- Array dimensions
- Pointers to the actual data
- A pointer to the complex part of the data (if any)
- For a structure or an object, the number of properties, member functions and their names

All variables used by MATLAB interpreter are stored in a container called "workspace". Variables need not to be declared in MATLAB. Once a variable is assigned a value (technically it is used as the left hand side argument), it appears in the workspace and the user can save its contents into a file using a proprietary binary format. The saved variables can be imported back into a workspace. Single variables or all variables can be deleted from a workspace via `clear` command.

3.2 MATLAB External Libraries

A MATLAB external library can be any dynamically shared object. The format of shared objects differ for different operating systems. MATLAB supports Executable and Linkable Format (ELF) on Unix-like systems, Portable Executable format (PE) on Microsoft Windows systems, and Mach Object file format (Mach-O) on Mac OS X systems. These formats are standard file formats for shared libraries and executables in the mentioned operating systems.

The dynamically shared libraries are loaded upon request and are mapped to the shared memory region. The shared memory region is a part of MATLAB virtual memory address space, where other shared libraries, other files and shared data are mapped. Besides, this region can effectively be used for inter-process communication as a non-kernel alternative to message passing systems [38, 26].

MATLAB custom shared libraries can be generated by a MATLAB command `mex` from source C/C++ codes containing symbols from MATLAB C/C++ API, called source MATLAB Executable files (source MEX-files). A source MEX-file is C/C++ source code, which defines specialized entry point – a function called `mexFunction`. This entry point function is declared in C header file `mex.h` distributed with MATLAB. This header file holds declarations of MATLAB C/C++ API functions and structures. MATLAB shared libraries generated by `mex` command are referred to as binary MATLAB Executable files (binary MEX-files). MATLAB uses proprietary file extensions for shared object generated by `mex` command[1].

Additionally, any C/C++ source code can include `mex.h` header file and use the declared functions. This source code can be compiled into a dynamically shared object and then imported into MATLAB via `loadlibrary` command. In other words, MATLAB C/C++ API is not limited to MATLAB MEX-files. Henceforth, the term MATLAB C/C++ API applies to both MEX-files and generic C/C++ files.

3.3 MATLAB Memory Allocation and Deallocation

This section describes necessary attributes of general C/C++ memory allocation/deallocation, specific MATLAB C/C++ API memory allocation/deallocation functions, and MATLAB garbage collector details.

C++11 standard (formerly known as C++0x) defines optional implementation of "transparent" garbage collection [29, 31]. However common C++ compilers either specify no support or ignore this proposal altogether [1, 5, 2]. C++ code often uses manual memory management in which case good memory management is founded on using good programming practices rather than explicit technology. The best common practice of dy-

1. `mex` command - documentation at
`http://www.mathworks.com/help/techdoc/ref/mex.html`

namic heap allocation in C and C++ is to keep consistency of memory ownership [19]. Each integral segment of dynamic heap memory is allocated by identifiable section or subroutine of a program. The same section or subroutine of the program should be responsible for deallocating this memory segment.

If memory is allocated on the heap inside a shared library, the relevant part of the shared library is responsible for memory deallocation. If the allocated memory is not properly deallocated by the library (*e.g.*, a routine is interrupted by the user or runs out of memory), a memory leak occurs. This memory is unavailable until MATLAB shuts down.

Unloading a library, which did not deallocate its memory[2], does not release the memory either. The memory remains allocated and it is even possible to reload the library, use the previously allocated memory and finally free the memory correctly if reference to the memory is kept (*e.g.*, in a MATLAB variable).

Both C/C++ and Fortran MATLAB API provide specialized memory allocation/deallocation functions `mxMalloc` and `mxFree`. `mxMalloc` is equivalent of C `malloc` function with the difference that the allocated memory is automatically deallocated while a binary MEX-file returns control to the MATLAB process. It is recommended to free the allocated memory at the end of a MEX-file by `mxFree` (equivalent to C free), because the manual clean-up is more efficient [11].

MATLAB is using exact reference counting to manage own heap allocated data. This is better thought of as a deterministic automatic memory management rather than advanced garbage collection [22]. As a consequence, features utilized by MATLAB such as copy-on-write mechanism [37] and heap deallocation on stack unwinding (on function return) [22] slow the computation down considerably.

On the other hand, such hidden language implementation details should not be exploited and misused when writing code in this language, because future implementation updates and fixes may result in speed penalty or even unexpected behavior. In software engineering hiding of the implementation details is referred to as the black-box principle [23]. If the details are hidden, there is no temptation to rely on them. The emphasis should be put on using appropriate data structures and algorithms.

2. Memory allocated by C++ `new` or C/C++ `malloc`, not by `mxMalloc`.

3.4 Memory Sharing and Persistence

All MATLAB variables in a workspace are stored in the shared memory region of the main MATLAB process. Therefore, the address space with all MATLAB variables can be used from within other library functions. However, the actual data is not accessible directly. Instead, MATLAB C/C++ and Fortran API [4] is used to get the access to its variables.

Memory allocated in a library function on the heap is accessible for other libraries and for MATLAB, until it is deallocated. The memory can be allocated via `malloc` function or C++ `new` operator. Memory and objects allocated in this way reside in memory until appropriate `free` or `delete` is issued.

It is possible to make the memory allocated with `mxMalloc` to persist different binary MEX-file calls using `mexMakeMemoryPersistent` function. It disables the automatic deallocation upon function completion. In such a case it is recommended to use function `mexAtExit` to register a function which deallocates memory used by a binary MEX-file. This registered function is executed when the binary MEX-file is removed from the MATLAB environment or MATLAB software terminates [10]. Apart from correct termination of a MEX-file these non-standard situations may occur:

- MEX-file exits by issuing error message to MATLAB (via API function `mexErrMsgTxt`[3])
- MEX-file calls API function `mexCallMATLAB` [4] to pass control to MATLAB and execute MATLAB function which creates an error
- MEX-file execution is interrupted by the user (Ctrl+C)
- MEX-file runs out of memory

In the first situation memory can be deallocated before issuing `mexErrMsgTxt`. In the second situation `mexCallMATLABWithTrap`[5] can be used in substitution to `mexCallMATLAB` with the option to capture an

3. This API function prints error message to MATLAB command window and aborts the MEX-file.
`http://www.mathworks.com/help/techdoc/apiref/mexerrmsgtxt.html`
4. Example usage: `mexCallMATLAB(0,NULL,1, x, "factorial");` which executes MATLAB function `factorial` with one parameter `x` (of type C struct `mxArray`).
Command documentation at:
`http://www.mathworks.com/help/techdoc/apiref/mexcallmatlab.html`
5. Command documentation at
`http://www.mathworks.com/help/techdoc/apiref/mexcallmatlabwithtrap.html`

error from MATLAB and return to the MEX-file and respond to it (in this case deallocate memory).

In the first two situations correct deallocation can be achieved using both standard allocation/deallocation functions and specialized MATLAB C++ API functions. In the latter two cases, however, the specialized MATLAB C++ API functions for memory management are superior to the standard ones, because if MATLAB C++ API memory allocation functions were used to allocate memory, automatic memory cleanup mechanism is raised up to avoid memory leaks [9].

The manual cleanup function `mexAtExit` mentioned above has limitations. It can only register a function of the following type: `void (* cleanup)(void)`. The cleanup function can therefore only access variables of the global scope, but massive usage of global variables is generally discouraged [43].

If third party shared library is used in MATLAB, it is best left responsible for its memory management. MATLAB C/C++ API allocation/deallocation functions provide no advantage to standard allocation/deallocation functions unless used with `mexAtExit` function. Our approach defaults to standard C/C++ memory management functions.

4 External MATLAB Interfaces

MATLAB provides interfaces to external routines written in various languages. C/C++ routines can be utilized by MATLAB Generic Shared Library interface, or by C++ MATLAB Executable files (C++ source code with predefined custom MATLAB entry point). Both possibilities are described and discussed later in this chapter. MATLAB also supports Java language code integration. This option is not considered in this thesis for the following reasons:

- Java does not support pointers, whereas MATLAB and C/C++ do.
- Java uses garbage collected memory management. Therefore another layer would be inserted into the design.
- The extra layer in Java based approach enforces extra type mapping of language features (MATLAB with Java, Java with C/C++).

4.1 MATLAB Generic Shared Library Interface

MATLAB provides functionality to dynamically load and use symbols from compiled shared ANSI C libraries called Generic Shared Library interface [8]. C++ libraries use name mangling to deal with overloading and namespaces [25]. Name mangling is compiler and platform specific and not standardized [24]. Therefore it is not directly and easily possible to call functions from shared C++ libraries via MATLAB Generic Shared Library interface. Appendix A gives an example of name mangling and our approach to it.

Standardized ANSI C libraries are compatible across platforms. The operating system takes the responsibility for the correct interpretation and execution of library functions (symbols in binary shared library). This does not force the library to be written in C programming language. However the functions in the library must have a C Application Binary Interface (ABI) [3]. This is ensured by a compiler.

MATLAB automatically maps primitive and some extended types [13]. A library must be modified to support other extended and user defined types. The MATLAB Generic Shared Library interface does not support function pointers as parameters of the library functions [8].

For ABI to work, Perl programming language and a supported C compiler must be available. Furthermore, during the initialization of the library,

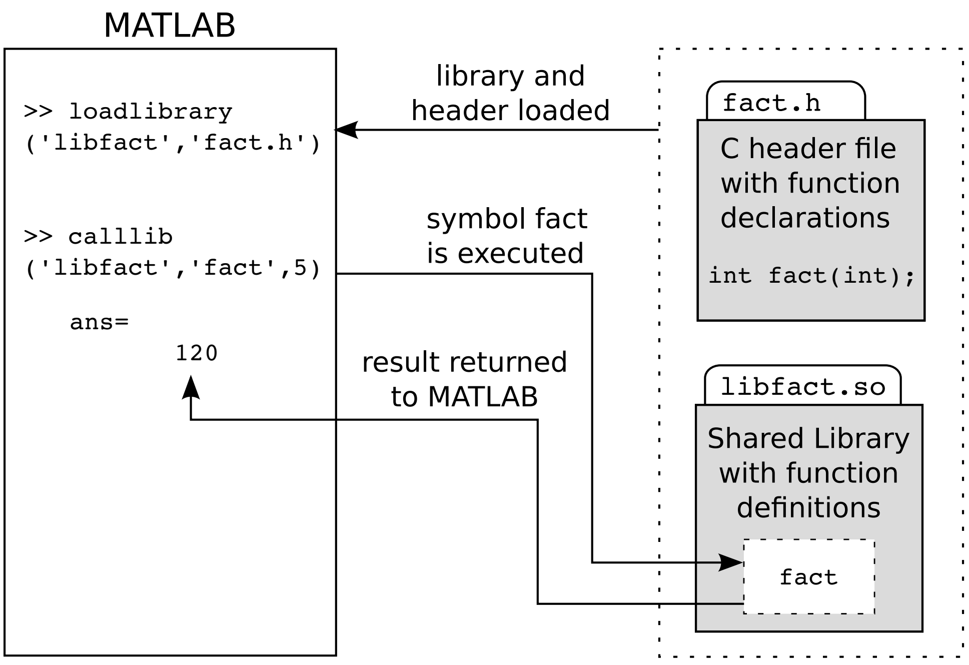


Figure 4.1: MATLAB Generic Shared Library interface operation example. MATLAB loads the shared library `libfact.so` along with header file `fact.h`. MATLAB parses the header file and looks for according symbols in the `libfact.so` library. Then `calllib` function executes symbol `fact` from the library and stores the result in `ans` variable and MATLAB prints it out to the command line.

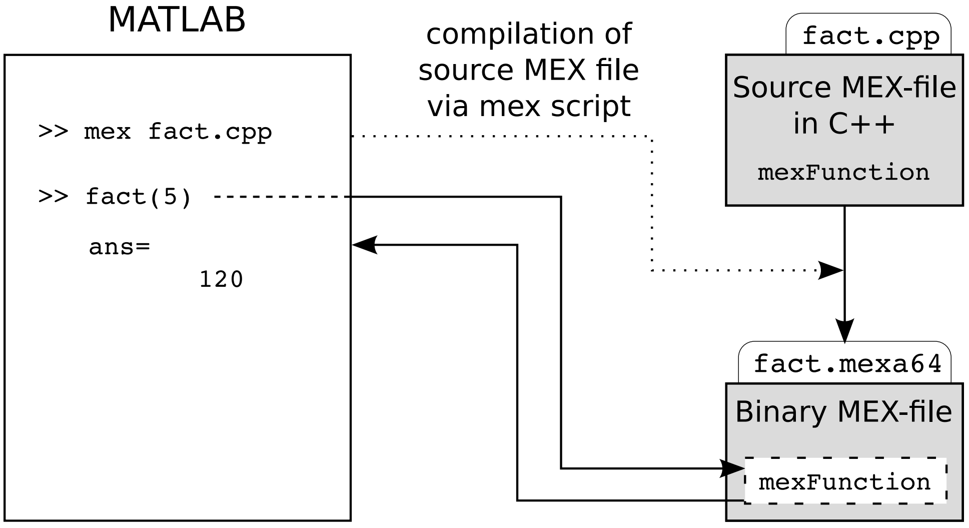


Figure 4.2: MEX-file compilation and operation example. First, source MEX-file `fact.cpp` is written, containing MATLAB C/C++ API `mexFunction`. The source MEX-file is compiled via the `mex` script into a binary MEX-file `fact.mexa64` (file extension for Linux). The binary MEX-file is executed by issuing the base of the filename (without file extension) on MATLAB command line.

a C header file with required function declarations must be provided. MATLAB compiler parses the header file and reads the information about function parameter and return types.

Figure 4.1 shows an example how a `fact` symbol can be loaded and executed from an external library. First, the library is loaded to into MATLAB by `loadlibrary` function. This command results in MATLAB parsing the input header file (`fact.h` in this case) and looking for the parsed function symbols in the shared library (`libfact.so` on Linux, `libfact.dll` on Microsoft Windows). Second, the `fact` symbol is executed by `calllib` function which passes number 5 as its argument. The result is returned to MATLAB workspace and stored in variable `ans`, which is printed to the screen in MATLAB. Finally, `unloadlibrary` can be issued to deallocate and unload the shared library from memory.

4.2 MATLAB Executable Files

External libraries can also be accessed from MATLAB via MEX-files (Section 3.2). Each source MEX-file contains an entry point called `mexFunction` . Its parameters determine the amount of parameters the function has been executed with in MATLAB as well as pointers to the data. MATLAB C++ API provides MATLAB data manipulation functions. Figure 4.2 illustrates a work flow with MEX-files.

Unlike shared libraries described in the previous section the source MEX-files can be written in C++ and the data conversion and manipulation is solely up to the programmer.

5 SWIG

This chapter defines necessary terminology used in the following chapters and illustrates SWIG role in wrapper code generation. Technically, SWIG is a dedicated C/C++ parser and syntax analyzer that parses C/C++ declarations and generates wrapper code [14, 18]. We define these terms[1]:

Source C++ Library is a C++ library that is desired to be interfaced into MATLAB. This library is provided by the user.

SWIG Interface File is a text file containing C++ function, object, and variable declarations and custom SWIG directives that influence the code generation [15].

C++ Wrapper Code is C++ source code generated by SWIG from an interface file.

C++ Library Interfaced to MATLAB is a dynamically shared library compiled from C++ wrapper code and source C++ library. This library is readily usable in MATLAB.

Wrapper User Interface is a collection of auxiliary files that simplify the usage of the C++ interfaced library in MATLAB interpreter.

MATLAB C++ Library Header File is a header file containing declarations of interface functions suitable to be used with MATLAB command `loadlibrary` as described in Section 4.1.

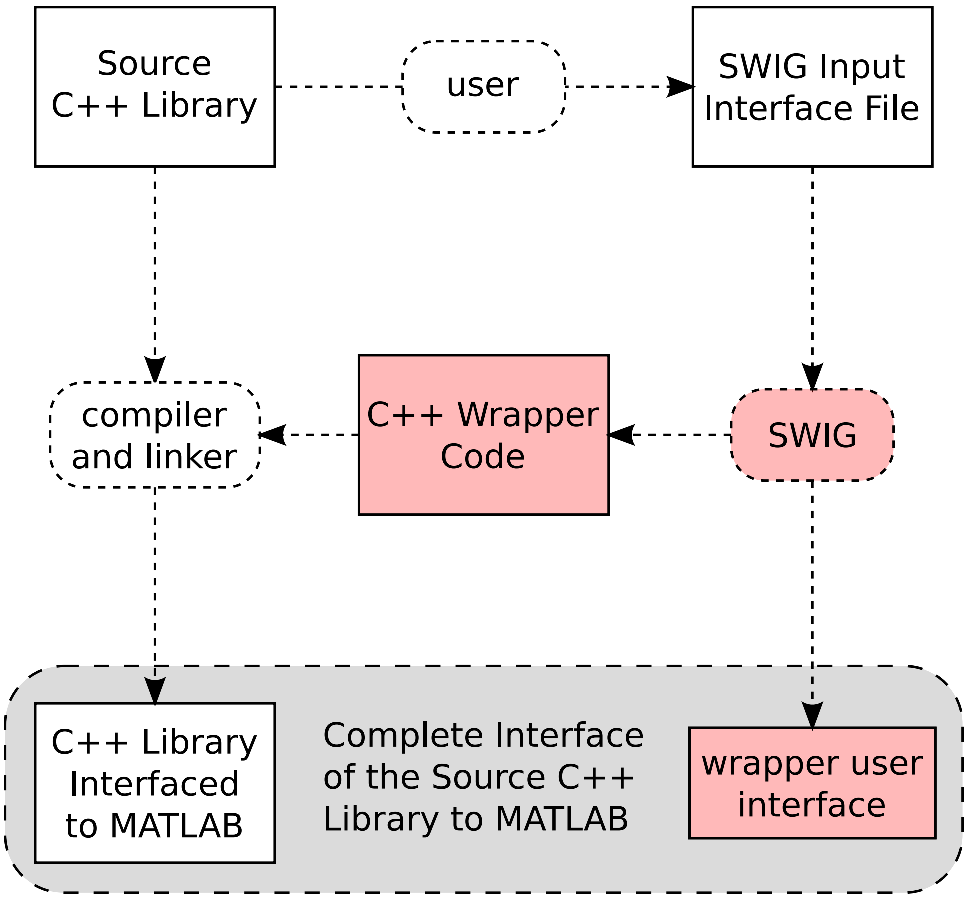
Figure 5.1 illustrates the work flow of producing a C++ library interfaced to MATLAB from the source C++ library. The work flow consists of the following:

1. The user identifies functionality that is exported to MATLAB.
2. Based on the previous step, the user creates a SWIG input interface file.
3. The user executes SWIG (specifying MATLAB module with `-matlab` switch) and passes the previously created interface file to SWIG as input.

1. Note that in this chapter we are using the term C++ in place of C/C++ to increase clarity of the document. In this context C can be considered a subset of C++.

4. SWIG produces C++ wrapper code. This code can be compiled and linked with the source C++ library.
5. SWIG also produces auxiliary files. In case of SWIG MATLAB module, SWIG produces a wrapper user interface and a MATLAB C++ library header file.
6. Source C++ library functions can be accessed via the generated wrapper user interface.

SWIG operation is independent of any third party components, *i.e.*, SWIG does not require MATLAB or the source C++ library to produce the C++ wrapper code. Likewise, the generated files are standalone and independent of SWIG, platform, and operating system.



Source C++ Library
user
SWIG Input Interface File
compiler and linker
C++ Wrapper Code
SWIG
C++ Library Interfaced to MATLAB
Complete Interface of the Source C++ Library to MATLAB
wrapper user interface

Figure 5.1: An overview of SWIG wrapper generation work flow. First, the user creates SWIG interface file containing function declarations of the source C++ library. SWIG processes the input interface file and produces C++ wrapper code and wrapper user interface in form of MATLAB class and function definitions. C++ wrapper code and source C++ library are compiled and linked into a C++ library interfaced to MATLAB. The produced library and the auxiliary files represent the complete solution (in grey) for interfacing the original source C++ library. Light red represents automated generation realized by our MATLAB SWIG module.

6 Wrapper Memory Management

Generated wrapper code must realize a layer between the source C++ library and MATLAB interpreter. From the point of view of memory management it should ensure that memory allocated by the source C++ library persists different library calls. In particular C++ objects created at MATLAB interpreter side must stay persistent between different C++ library calls and stay synchronized with their MATLAB counterparts.

In this chapter, possible data persistence and synchronization schemes are described and discussed.

The ultimate goal is to enable automatic interfacing of C++ libraries into MATLAB. The described data persistence and synchronization schemes are discussed with respect to the following criteria, which are considered to correspond to the optimal wrapper behavior:

- The generated library interface should manage memory efficiently
- Wrapper code generation should be as effortless as possible
- Wrapper user interface of the wrapped library in MATLAB should be user friendly
- Utilization of the wrapped library should not require advanced knowledge of compiling and linking

6.1 Common Features

Some ideas are common to all persistence and synchronization schemes. This section explains general issues that need to be addressed in order to produce a working wrapper for a generic C++ library. Language features found in C++ libraries include:

- C/C++ functions, C++ operators
- C++ polymorphism, templates, namespaces, streams
- Primitive data types (constant, local and global variables)
- C structures
- C++ classes (often passed by references)
- C++ Standard Template Library (STL) entities

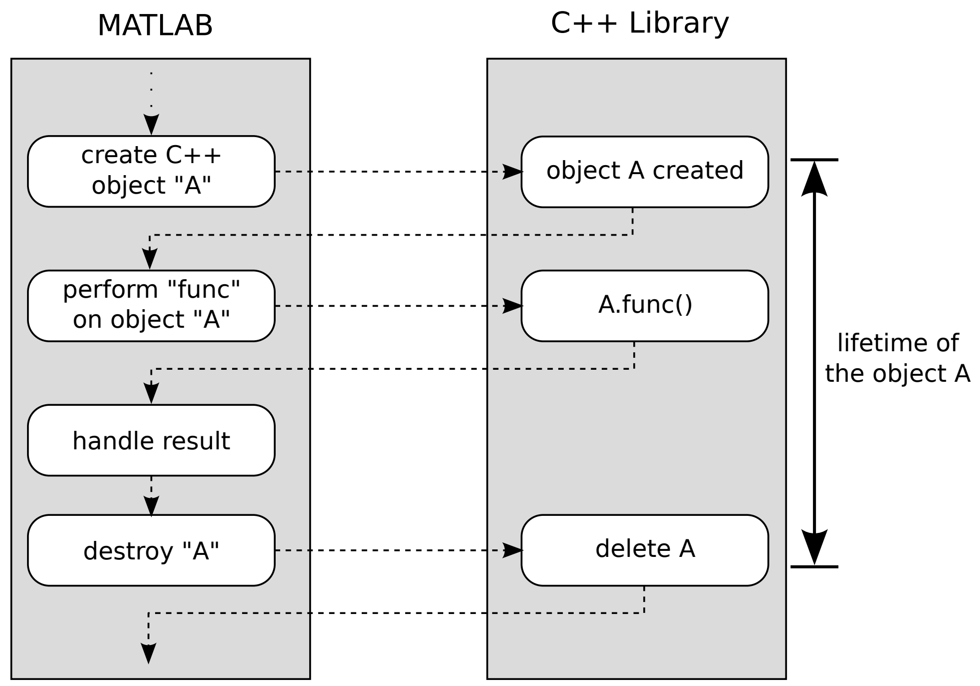


Figure 6.1: Control flow passing between MATLAB and C++ library and the lifetime of a heap allocated object spanning across multiple MATLAB and C++ operations.The term *lifetime of an object* here is conceptual and refers to the time span in which the object exists in *some* form.

Although this is not an exhaustive list, those are the central features C++ libraries often use [39, 33, 17]. For the SWIG module to be generic it has to take all of these features into account.

From the point of view of memory management the most important features are those where new objects/variables are created, deleted, or modified. Mapping of different language features is detailed in Jan Urban's thesis [41].

In contrast to the first two entries in the list above the latter four represent entities that need to stay persistent among different library calls, if they are allocated on the heap. As shown in Figure 6.1 different C++ library interactions (calls) may pass flow control back and forth between MATLAB and C++. Further, library calls may alter the memory.

6.2 Data Persistence and Synchronization Schemes

Generally there are three conceivable schemes to address persistence of C/C++ entities and their synchronization with their MATLAB counterparts. The first one uses an idea, that all data are transferred and converted to data of a newly created MATLAB object. The second strategy is to represent newly created C++ object by a handle object [7] on the MATLAB side. The handle object in MATLAB holds only a reference or a pointer to the C++ object. The last strategy may combine the two, letting the user decide which data to copy and which to reference. Figure 6.1 applies to all three cases.

6.2.1 Copy All to MATLAB

The first scheme assumes all C++ data to be copied back to MATLAB when a C++ subroutine ends. This scheme is illustrated in Figure 6.2[1] . For this scheme to be realized, either the shared library must define functions, which convert and transfer the data to and from MATLAB, or a generic conversion independent of data type would have to be used. In the former case, the conversion and transfer functions would produce intelligible data on MATLAB side. This data could then be further processed in MATLAB. In the latter case, the generic conversion could at best generate serialized[2] data, which would need extra functionality specific to the library (parsing and interpretation of the serialized data) to be useful in MATLAB, hence would require further development effort.

1. All examples in this thesis use MATLAB language's dot notation rather than function notation.
`http://www.mathworks.com/help/techdoc/matlab_prog/bruka7z.html#brukbf9`
2. Serialization is a transformation of data (usually an object) into one-dimensional stream of bits. Serialization is often used to save complex data types to a hard disk or to send over network [21].

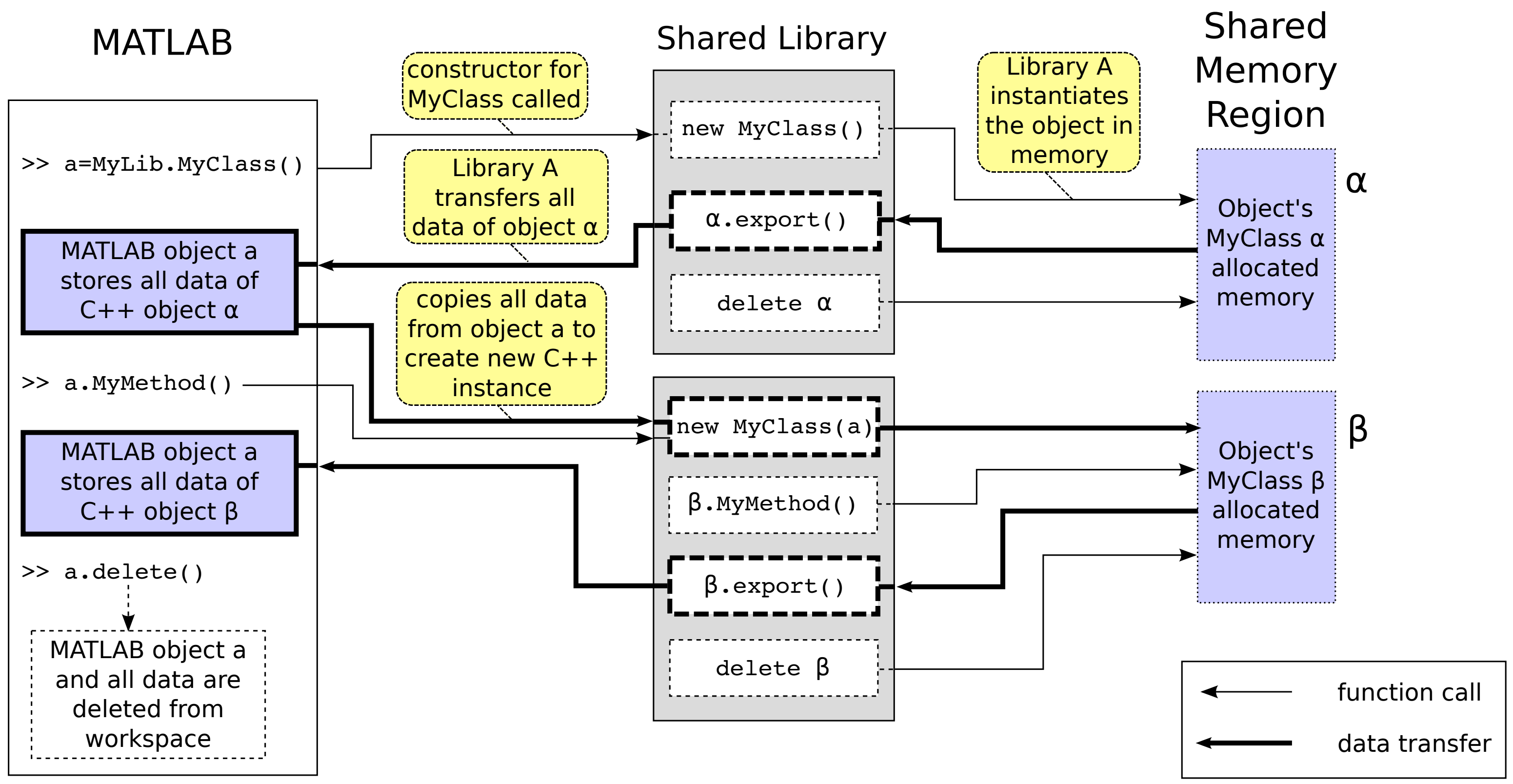

MATLAB
Shared Library
Shared Memory Region
>> a=MyLib.MyClass()
constructor for MyClass called
new MyClass()
Library A instantiates the object in memory
α
Object's MyClass α allocated memory
Library A transfers all data of object α
α.export()
MATLAB object a stores all data of C++ object α
delete α
copies all data from object a to create new C++ instance
>> a.MyMethod()
new MyClass(a)
β
Object's MyClass β allocated memory
MATLAB object a stores all data of C++ object β
β.MyMethod()
β.export()
>> a.delete()
MATLAB object a and all data are deleted from workspace
delete β
function call
data transfer

Figure 6.2: Illustration of copying all data of a C++ object. MATLAB object `a` stores all data of the C++ object α. Object α is deleted after exporting the data to MATLAB. Invocation of function `MyMethod` on object `a` results in copying all the data, passing it into dedicated constructor, and creating C++ object β. Function `MyMethod` is carried out on object β, the data is imported back to MATLAB and object β is deleted. Deletion of object `a` deletes the data it stores.

SWIG only processes syntactic information and cannot interpret function definitions and their semantics. Consequently, SWIG cannot produce data conversions and transfer routines for a general library from principle.

Memory management in this scheme is simple as allocated C++ objects can be deallocated on routine exit. Further, the objects can be allocated on the stack rather than the free store memory segment[3] to relax the necessity of deallocation. This scheme may introduce overheads because memory allocations/deallocations could happen on each C++ library function call.

If a C++ object itself allocates heap memory, then even the generic serialization may not deliver expected results as it would transfer only the actual memory of the object and not the extra allocated memory.

6.2.2 Reference C++ Object to MATLAB

In the second scheme all library data stored in memory are referenced from MATLAB. Figure 6.3 illustrates that when the object `a` is created in MATLAB, the interface instantiates the counterpart C++ object and reports the reference back to MATLAB object `a`, which saves it. This approach has little overhead as the only data transfered are references (memory pointers).

The lifetime of a C++ object (α in the example) corresponds to the lifetime of its MATLAB counterpart (object `a`).

In this case the data of the C++ object are not readily present in MATLAB. However, if the library provides its own general conversion and transfer functions[4], the data can be transfered on user demand.

3. Free store is a memory segment used by C++ to store objects allocated with the `new` operator. Free store is often referred to as heap, because many implementations (GNU Compiler Collection [6]) use the same memory segment and, moreover, method of allocation [39, 6].
4. Functions, which are not MATLAB specific. For example C function `void fillData(double *fillArray)`, whose pointer parameter is interpreted as an array.

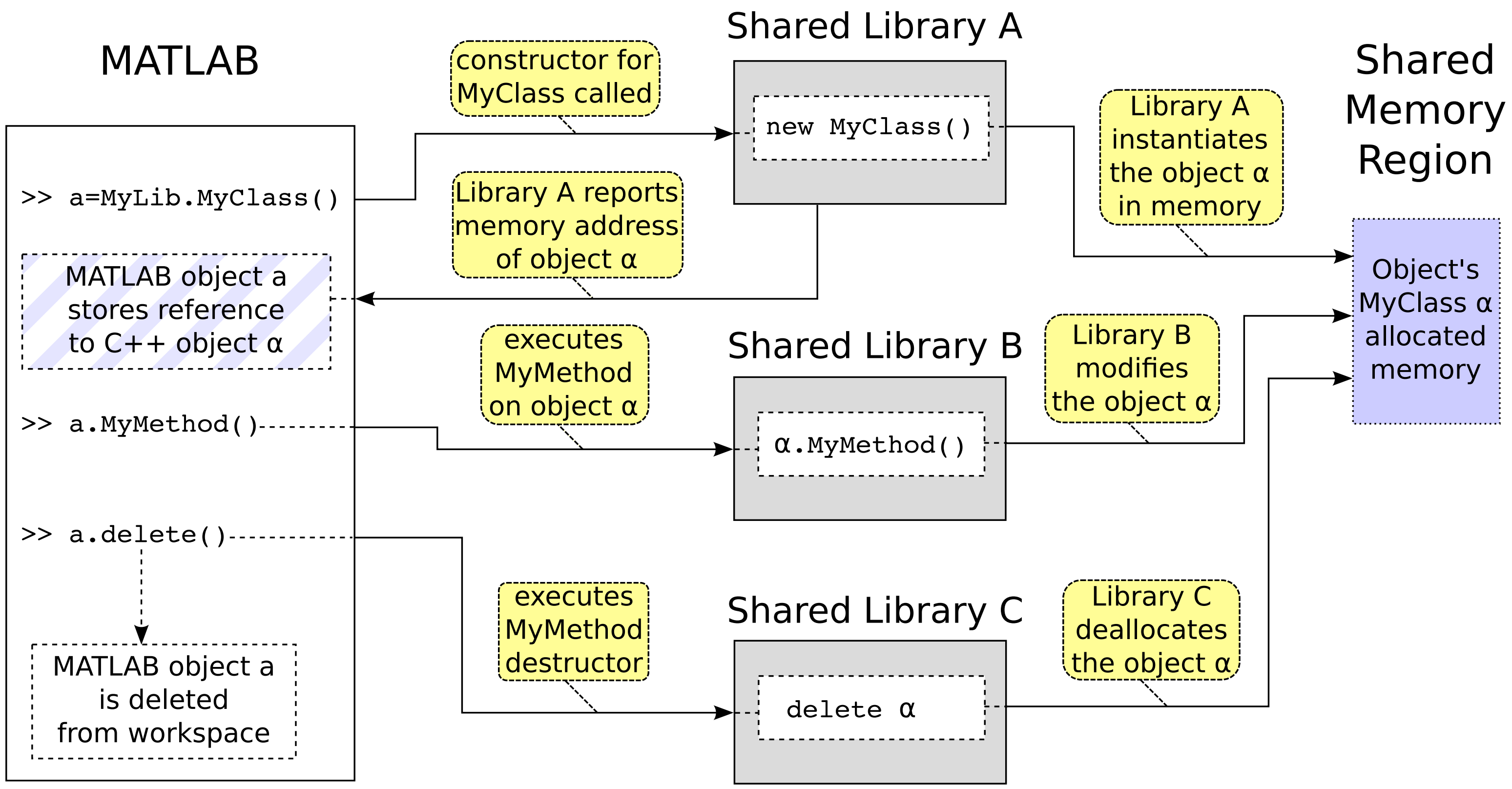

MATLAB
>> a=MyLib.MyClass()
MATLAB object a stores reference to C++ object α
>> a.MyMethod()
>> a.delete()
MATLAB object a is deleted from workspace
constructor for MyClass called
Library A reports memory address of object α
executes MyMethod on object α
executes MyMethod destructor
Shared Library A
new MyClass()
Shared Library B
α.MyMethod()
Shared Library C
delete α
Library A instantiates the object α in memory
Library B modifies the object α
Library C deallocates the object α
Shared Memory Region
Object's MyClass α allocated memory

Figure 6.3: Illustration of referencing the created C++ object by a handle class object. MATLAB object a stores reference to a C++ object α. Memory of the C++ object α remains persistent until explicitly deallocated (triggered by `a.delete()` in MATLAB) as described in Section 3.4. Note that the instantiated object's memory can be accessed from different shared libraries as it is stored in the shared memory region.

6.2.3 User Defined - Combination of Above

Copying some data and referencing the rest is the last option. It may be advantageous to have the user decide on which data to transfer and which to reference. Similarly as in both previous schemes, custom conversion and transfer routines must be provided to produce legible data in MATLAB. Therefore, in principle this option does not differ from referencing (6.2.2) as the user may use the transfer routines manually.

6.3 Data Persistence Scheme in Our Implementation

The SWIG MATLAB module that we developed uses referencing of all C++ data described in 6.2.2. This approach is superior to the others, because it provides:

- Little data transfer, little overhead
- In case the source C++ library provides custom data conversion and transfer, referencing scheme can utilize these functions just as the other schemes
- No unnecessary data allocations/deallocations
- Simple mental model

In this scheme each C++ object α has its MATLAB counterpart object `a`. Object `a` inherits MATLAB class *handle*, which represents a reference to an outside resource. Object `a` has one property named *pointer* of MATLAB type `libpointer`[5], which holds a reference to the C++ object α. MATLAB type `libpointer` abstracts from memory address pointer and accurately simulates memory pointer on each platform, removing possible ambiguity of memory address interpretation, which could occur among systems with different endianness[6].

5. `http://www.mathworks.com/help/techdoc/ref/libpointer.html`
6. `http://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Data/endian.html`

7 Wrapper External Interface Options

There are three drafts of wrapper design regarding the choice of external interface. The first option is that the SWIG MATLAB module generates one MEX-file, that interfaces the whole source C++ library. The second is to use many MEX-files - one MEX-file per each required atomic functionality[1]. The third option is to generate one C++ source file, which holds wrapper functions declared as `extern "C"`. This file is then compiled and linked with the source C++ library into a shared library, which can be loaded by MATLAB Generic Shared Library interface.

7.1 Single MEX-file

This approach uses an idea, that the whole interface code is contained in one C++ source MEX-file. This MEX-file has single point entry function `mexFunction`, which decides on what library routine to execute on the basis of received parameters. One of the parameters is the name of the desired library function. Figure 7.1 illustrates this idea.

7.2 Multiple MEX-files

This strategy assumes that each library function is wrapped by one MEX-file. As the name of the entry point function in a source MEX-file is fixed to `mexFunction`, each MEX-file must be compiled into a shared library separately to avoid symbol name conflict, resulting in code bloat[2]. Due to this each MEX-file has to dynamically link the source C++ library containing function definitions. This strategy is illustrated in Figure 7.2.

7.3 MATLAB Generic Shared Library Interface Wrap

Each function is wrapped by one function in the generated C++ wrapper code. For C++ functions, corresponding ANSI C functions are generated. The features of C++ that require name mangling in object files are wrapped by multiple ANSI C functions, which effectively translate the C++ ABI name mangling to source code. This way, the module produces one

1. Each function overload, each template specified function, each member function, get and set functions for global and public member variables.
2. `http://docforge.com/wiki/Code_bloat`

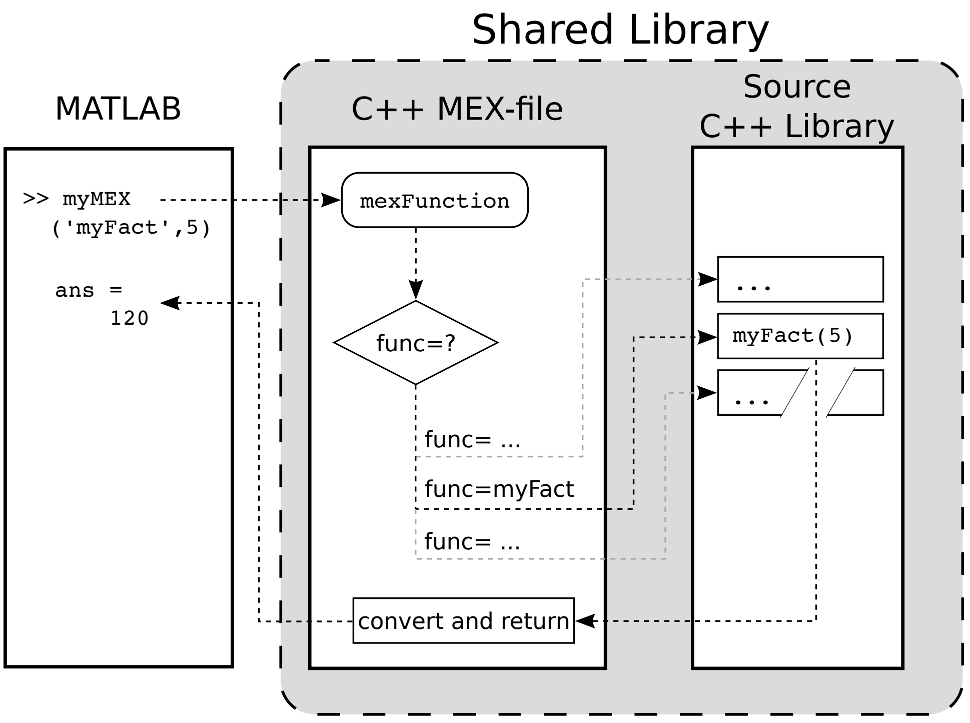


Figure 7.1: Single MEX-file. Executing `myMEX` in MATLAB executes the binary MEX-file. The entry point function receives the name of the function and parameters. The entry point function executes corresponding symbol from the library. The result is returned to MATLAB. This figure indicates that the MEX-file together with the library form one MATLAB interfaced shared library.

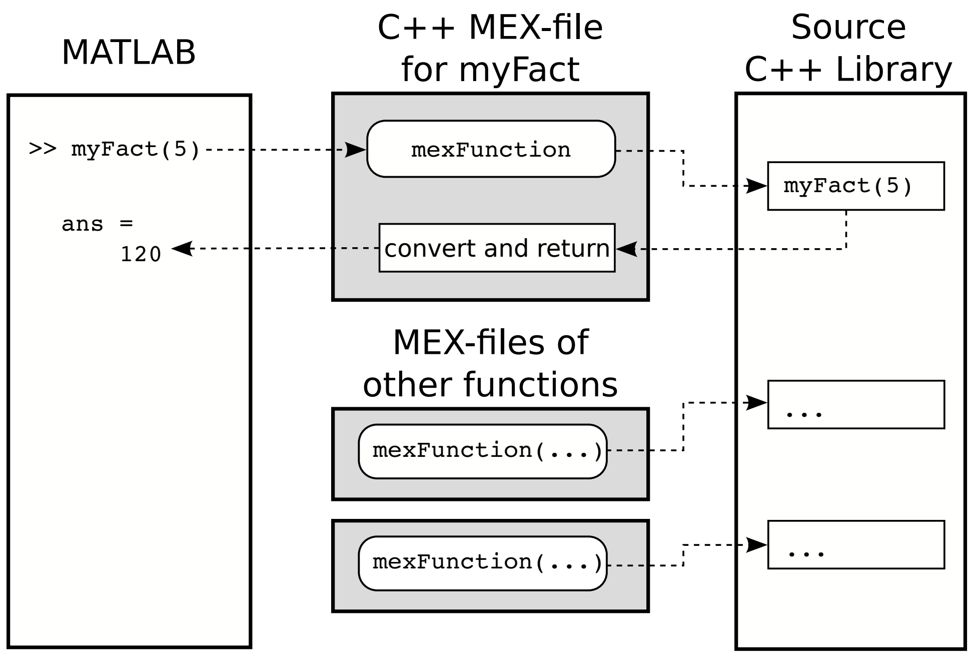


Figure 7.2: Multiple MEX-file design. Each function from the source C++ library corresponds to and is interfaced by one MEX-file. Each MEX-file dynamically links source C++ library.

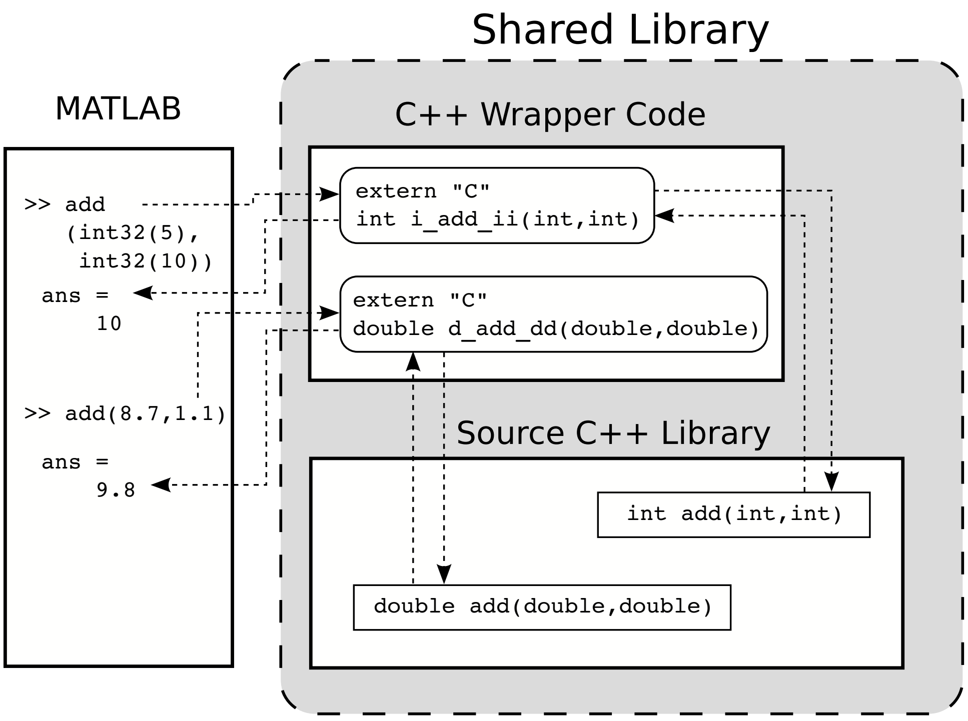

Shared Library
MATLAB
C++ Wrapper Code
>> add (int32(5), int32(10))
ans = 10
>> add(8.7,1.1)
ans = 9.8
extern "C" int i_add_ii(int,int)
extern "C" double d_add_dd(double,double)
Source C++ Library
int add(int,int)
double add(double,double)

Figure 7.3: Single generic C++ shared library. The source C++ library defines overloaded function `add` for `double` and `int` types. This results in name mangling in binary representation. The C++ wrapper code defines two functions with different (mangled) names. Each calls the corresponding overloaded function from the C++ library. It is the wrapper user interface that hides the source code-level name mangling from the user in MATLAB (not displayed for simplicity).

C++ wrapper source file with `extern "C"` functions. Figure 7.3 illustrates the principle.

We selected MATLAB Generic Shared Library interface wrapper design (Section 7.3) as the external interface scheme for our solution. The reasons that support this choice are:

- Single C++ wrapper output file
- No code bloat
- MATLAB Generic Shared Library interface provides conversions of primitive C types

8 Conclusion

This thesis concerned memory management issues related to automatic wrapper code generation needed to interface C++ libraries into MATLAB using SWIG.

The text analyzes memory management of MATLAB and its external interfaces considering memory sharing between components. Feasible designs of memory sharing and persistence were discussed and compared. Chosen design details were described. Based on analysis of memory management of MATLAB and MATLAB external libraries (Chapter 3) we proposed reasonable schemes of addressing external library data persistence (Chapter 6). From the provided options of interfacing external libraries to MATLAB we examined MEX-files and MATLAB Generic Shared Library interface (Chapter 4). Based on the findings we proposed three options of wrapper design (Chapter 7). With automated generation by SWIG in mind, we chose and combined one data persistence scheme with a strategy to interface external libraries to design a viable wrapping approach.

The wrapping approach described in this thesis has been implemented in a SWIG MATLAB module that Jan and I developed and is part of this thesis. The module generates wrapper code that interfaces C++ objects persistently, which means that C++ objects can be used from MATLAB command line.

The developed module[1] is fully functional in a sense it can successfully generate wrapping code for general libraries that contain the following features: all primitive data types including C structures, C functions, C++ classes, C++ function overloading, namespace scope functions, template functions. However, there are some C++ features which are not fully supported by our module: template classes, namespace scope classes, and operator overloading. These features were not required in the official assignment of the thesis and are compatible with the proposed wrapping approach.

In future, we want to add function pointer support, Standard Template Library support, exception handling support, and documentation integration, which are the C++ features required for a module to be included in the standard SWIG distribution.

1. The module is available at `https://github.com/twiho/Matlab-in-SWIG`.

References

[1] C++0x/C++11 Support in GCC. http://gcc.gnu.org/onlinedocs/libstdc++/manual/bk01pt04ch11.html (ref. 19th May 2012).

[2] C++98 and C++11 Support in Clang. http://clang.llvm.org/cxx_status.html#cxx0x (ref. 23th May 2012).

[3] Calling Functions in Shared Libraries - online Mathworks Matlab Product Documentation. http://www.mathworks.com/help/techdoc/matlab_external/f43202.html (ref. 25th May 2012).

[4] C/C++ and Fortran API Reference - online Mathworks Matlab Product Documentation. http://www.mathworks.com/help/techdoc/apiref/bqoqnz0.html (ref. 25th May 2012).

[5] DMC++ Support for C++0x (C++11). http://www.digitalmars.com/ctg/CPP0x-Language-Implementation.html (ref. 25th May 2012).

[6] GNU Online Documentation - Memory. http://gcc.gnu.org/onlinedocs/libstdc++/manual/bk01pt04ch11.html (ref. 19th May 2012).

[7] handle - online Mathworks Matlab Product Documentation. http://www.mathworks.com/help/techdoc/ref/handle.html (ref. 25th May 2012).

[8] loadlibrary - online Mathworks Matlab Product Documentation. http://www.mathworks.com/help/techdoc/ref/loadlibrary.html (ref. 25th May 2012).

[9] Memory Management - online Mathworks Matlab Product Documentation. http://www.mathworks.com/help/techdoc/matlab_external/f25255.html (ref. 22nd May 2012).

[10] mexAtExit - online Mathworks Matlab Product Documentation. http://www.mathworks.com/help/techdoc/apiref/mexatexit.html (ref. 25th May 2012).

[11] mxMalloc - online Mathworks Matlab Product Documentation. http://www.mathworks.com/help/techdoc/apiref/mxmalloc.html (ref. 25th May 2012).

[12] online Mathworks Matlab Product Documentation. http://www.mathworks.com/help/techdoc/ (ref. 22nd May 2012).

[13] Passing Arguments to Shared Library Functions. http://www.mathworks.com/help/techdoc/matlab_external/f42387.html(ref. 17th May 2012).

[14] Simplified Wrapper and Interface Generator. http://www.swig.org/ (ref. 25th May 2012).

[15] SWIG Directives - SWIG Online Documentation. http://swig.org/Doc2.0/SWIG.html#SWIG_nn7 (ref. 25th May 2012).

[16] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Professional, 2001.

[17] Matthew H. Austern. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Addison-Wesley Professional, 1999.

[18] David M. Beazley. *SWIG: an easy to use tool for integrating scripting languages with C and C++*. TCLTK'96. USENIX Association, Berkeley, CA, USA, 1996.

[19] George Belotsky. C++ Memory Management: From Fear to Triumph, Part 2. 2003.

[20] Albin M. Butters. *ACM/IFIP/USENIX International Middleware Conference Companion - Total cost of ownership - A comparison of C/C++ and Java, December 1-5, 2008, Leuven, Belgium*. Association for Computing Machinery, New York, N.Y, 2008.

[21] Marshall Cline. Serialization and Unserialization. http://www.parashift.com/c++-faq-lite/serialization.html (ref. 25th May 2012), 1991-2011.

[22] Dave Foti. Inside MATLAB Objects in R2008a, MATLAB developer blog. http://www.mathworks.com/company/newsletters/digest/-2008/sept/matlab-objects.html, 2008. (ref. 25th May 2012).

[23] David Drysdale. *High-Quality Software Engineering*. Lulu.com, 2007.

[24] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley Professional, 1990.

[25] Agner Fog. Calling conventions for different C++ compilers and operating systems. `http://www.agner.org/optimize/calling_conventions.pdf` (ref. 17th May 2012), 2004-2012.

[26] Greg Gagne, Peter B. Galvin, Peter Galvin, and Avi Silberschatz. *Applied Operating System Concepts*. Wiley, 1999.

[27] Mel Gorman. *Understanding the Linux Virtual Memory Manager*. Prentice Hall, 2004.

[28] M. Gregoire, N. Solter, and S. Kleper. *Professional C++*. John Wiley & Sons, 2011.

[29] Mike Spertus Hans J. Boehm. *ISMM '09 : Proceedings of the 2009 ACM SIGPLAN International Symposium on Memory Management : June 19-20, 2009, Dublin, Ireland, Garbage Collection in the Next C++ Standard*. Association for Computer Machinery, New York, N.Y, 2009.

[30] Matthew Hertz. Quantifying the performance of garbage collection vs. explicit memory management. In *in: Proc. ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA*, pages 313–326. ACM Press, 2005.

[31] International Organization for Standardization. ISO/IEC 14882:2011 - C++ Standard. `http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?csnumber=50372&ICS1=35&ICS2=060` (ref. 21st May 2012).

[32] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management (Chapman & Hall/CRC Applied Algorithms and Data Structures series)*. Chapman and Hall/CRC, 2011.

[33] Brian W. Kernighan and Dennis M. Ritchie. *C Programming Language (2nd Edition)*. Prentice Hall, 1988.

[34] S. Lipschutz. *Schaum's outline of theory and problems of data structures:*. Série Schaum. McGraw-Hill, 1986.

[35] Josef Pacula. Connecting Morph-M into Matlab using SWIG. [Bachelor thesis] Faculty of Informatics, Masaryk University, Brno, 2010.

[36] Andor Pathó. Connecting i3dlib Library to Matlab. [Bachelor thesis] Faculty of Informatics, Masaryk University, Brno, 2007.

[37] Loren Shure. Memory Management for Functions and Variables. `http://blogs.mathworks.com/loren/2006/05/10/memory-management-for-functions-and-variables`(ref. 25th May 2012), 2006.

[38] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. Wiley, 2008.

[39] Bjarne Stroustrup. *The C++ Programming Language: Special Edition*. Addison-Wesley Professional, 2000.

[40] Jan Urban. Connecting Morph-M into Matlab using SWIG. [Bachelor thesis] Faculty of Informatics, Masaryk University, Brno, 2010.

[41] Jan Urban. Interfacing C++ libraries to Matlab. [Master thesis] Faculty of Informatics, Masaryk University, Brno, 2012.

[42] Paul R Wilson. Uniprocessor garbage collection techniques. *Memory Management*, 49(September):1–42, 1992.

[43] W. Wulf and Mary Shaw. Global variable considered harmful. *SIGPLAN Not.*, 8(2):28–34, February 1973.

A C++ name mangling

C++ name mangling is what prevents the MATLAB Generic Shared Library interface to be compatible with C++ shared libraries. The main idea with this wrapper design is a translation of the reason for C++ name mangling to source code level. Two overloaded functions

```
int add(int, int);
double add(double, double);
```

need to be mangled in a external library, as function overloading is not part of "ANSI C" standard. GNU Compiler[1] produces the following symbols:

```
_Z3addii
_Z4add2dd
```

Our design responds to this situation by defining wrapping functions with mangled names on the source code level using `extern "C"` directive that tells the compiler not to mangle the resulting symbol. Of course this is possible only if names of the functions are different, otherwise name conflict would occur. The C++ wrapper code defines functions

```
extern "C" int add_SWIG0(int, int);
extern "C" double add_SWIG1(double, double);
```

which wrap around the original functions. Compilation of the C++ wrapper produces object file with symbols which are usable with MATLAB Generic Shared Library interface:

```
add_SWIG0
add_SWIG1
```

1. Standard Linux compiler, `http://gcc.gnu.org/`.