

C Stored Function Framework



12301 Research Blvd.
Building IV, Suite 410
Austin, TX 78759

U.S. Help Desk Phone: +1-800-546-9646 (or direct +1-512-697-3000), select ext. 3400

U.K. Help Desk Free Phone: 0800 032 6063

Europe Help Desk Phone: +44 20 7190 2947

Help Desk Email: support@lim.com

+1-512-697-3001 (Fax)

Part Number: 083_84

Date: March 6, 2008

Copyright © 2005-2008 by Logical Information Machines, Inc.

Patented May, 1995 U.S. Patent No. 08/392, 612

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of Logical Information Machines, Inc.

Restricted Rights Legend

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subdivision (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at 252.227-7013.

Logical Information Machines, Inc.
120 North LaSalle Street
Suite 2150
Chicago, IL 60602

(312) 456-3000

Product names mentioned herein are for identification purposes only and may be trademarks and/or registered trademarks of their respective companies.

While every precaution has been taken in the preparation of this manual, we assume no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information contained herein. Logical Information Machines, Inc. may revise this publication from time to time without notice.

Table of Contents

C Stored Function Framework	1
Description	1
Structure of a C Stored Function	2
Stored Function Framework	5
Regular Stored Functions	5
Built-in Stored Functions	10
Stored Function Life Cycle	12
Example: Providing a C Stored Function Wrapper for an Existing C Function	13
Straightforward Approach	13
Lazy Evaluation Approach	19
Complete Predictor Code	21
Complete Lazy Predictor Code	25
Index	29

C Stored Function Framework

Description

The purpose of the C stored function framework (framework, for short) is to allow easy introduction of new functionality into the MIM server. The main advantage of the code written within the C stored function framework is that the stored function implementation can access any of the internal MIM objects (such as `schSchema` object, for example).

New functionality written within the framework can be called in the "usual" way, using the following `bmim_client` query, for example:

```
%exec.units: 1 hour

SHOW
  1: slice (10, 15, 2004)
WHEN
  Date is after 1/1/2004
AND
  Date is Wednesday
```

where `slice` is the name of the C stored function.

This document is structured as follows.

- [“Structure of a C Stored Function”](#) describes the components of C stored functions.
- [“Stored Function Framework”](#) describes the utility functions available to writers of C stored functions. These utility functions expose part of the functionality of the MIM server.
- [“Stored Function Life Cycle”](#) describes the way the MIM server interacts with a C stored function while executing a user query.
- [“Example: Providing a C Stored Function Wrapper for an Existing C Function”](#) illustrates a complete example. Readers may wish to skim over the material in [“Structure of a C Stored Function”](#) through [“Stored Function Life Cycle”](#), and then review that material after reading [“Example: Providing a C Stored Function Wrapper for an Existing C Function”](#).
- [“Complete Predictor Code”](#) and [“Complete Lazy Predictor Code”](#) contain all the source code for the examples.

Structure of a C Stored Function

Each new set of code should be implemented as a C file, which we refer to as a C stored function. Since each stored function will eventually become a library in which the MIM server will load upon startup, the C implementation was chosen so as to avoid problems with loading C++ libraries.

It is essential for a stored function to be able to maintain a state, which would persist between invocations of different stored function functions. The stored function state is represented by the following structure (which we call stored function arguments, thus the name):

```
typedef struct {
    //error string
    char                *error;
    //pointer to a SchSchema object
    void                *schema;
    //pointer to an ExXmimOptions object, containing user defaults
    const void          *user_defaults;
    //pointer to an ExCompiledQuery object corresponding to the current query
    void                *compiled_query;
    //MimUnits (ExRelcol units)
    int                 units;
    //ExRelcol modulus
    int                 modulus;

    //pointer to the cfunc's characteristic time series,
    //a time series which has non-NaN values for times
    //when this c stored function is defined and NaN
    //values when it is not
    void                *characteristic_time_series;
    //pointer to the cfunc's ExTradingPattern object
    void                *trading_pattern;
    //pointer to the cfunc's ExDateRange object
    void                *trading_date_range;
    //cfunc specific state
    xmim_cfunc_state    *state;
    //constant int args
    xmim_cfunc_int_args *int_args;
    //constant double args
    xmim_cfunc_double_args *double_args;
    //category args
    xmim_cfunc_         *category_args;
    //string args
    xmim_cfunc_string_args *string_args;
    //ATTR args
    xmim_cfunc_attr_args  *attr_args;
    //time offset args
    xmim_cfunc_time_offset_args *offset_args;
    //period args
    xmim_cfunc_time_period_args *period_args;
    //security args
    xmim_cfunc_security_args  *security_args;
    //default return value
    double                    default_val;
    //return value of the get_value method
    double                    result;
} xmim_cfunc_args;
```

This structure has a member called `state`, which is an array of `void*`, specific to a particular stored function. A stored function may store any information it needs to persist between the function calls in this array.

Each stored function has the following structure. All functions are mandatory, unless specified otherwise. The function `cfunc_instance_init_defaults` initializes the default values of the stored function. Each stored function should "know" about the arguments it expects. The stored function will initialize the appropriate arguments in this function, i.e., assign the values to the names and size members of the appropriate structs.

```
extern "C" int cfunc_instance_init_defaults (xmim_cfunc_args *args) {
    ...
}
```

For example, if a stored function expects to get three constants as arguments, it should do something analogous to the following in the `cfunc_instance_init_defaults` function:

Also, we suggest that a stored function initialize `args->state` in this function.



This can be performed in the `cfunc_instance_init` function as well.

```
int num = 3;
args->int_args->size = num;
args->int_args->ptr = new int[num];
for (int i = 0; i < num; i++)
    args->int_args->ptr[i] = 0;

args->int_args->names = new char*[num];
args->int_args->names[0] = "day";
args->int_args->names[1] = "month";
args->int_args->names[2] = "year";
```

The function `cfunc_instance_init` initializes the state (`argsstate`) of a stored function. Everything that needs to be initialized before the `cfunc_instance_get_value` function will be called should be initialized here.



The stored function's input parameters are *set* at this point and can be used by the stored function to initialize its state if necessary.

```
extern "C" int cfunc_instance_init (xmim_cfunc_args *args) {
    ...
}
```

The optional function `cfunc_instance_compute_trading_pattern` computes the trading pattern for a cfunc.

```
extern "C" int cfunc_instance_compute_trading_pattern (xmim_cfunc_args *args) {
    ...
}
```

The optional function `cfunc_instance_compute_trading_date_range` computes the trading date range for a stored function.

```
extern "C" int cfunc_instance_compute_trading_date_range (xmim_cfunc_args *args) {
    ...
}
```

The optional function `cfunc_instance_get_characteristic_time_series` computes the characteristic time series for a stored function. A characteristic time series is a time series which has non-NaN values for times when the date exists and NaN values for times when the date does not exist.

```
extern "C" int cfunc_instance_get_characteristic_time_series (xmim_cfunc_args *args) {
...
}
```

The function `cfunc_instance_get_value` sets `args->result` to the value this stored function needs to return for the date and time `dt`.

```
extern "C" int cfunc_instance_get_value (xmim_cfunc_args *args, xmim_cfunc_date_time *dt) {
...
}
```

The function `cfunc_instance_destroy` is a destructor for a stored function. A stored function should destroy all the instances it creates.

```
extern "C" int cfunc_instance_destroy (xmim_cfunc_args *args) {
...
}
```

C Stored Functions can have constant (integer, double), attribute, time offset, time period, security, category, and string input arguments. The first five argument types are the same as arguments accepted by macros. The category argument is stored as a string and can be used to retrieve and process all relations under that category in the MIM schema hierarchy. The string argument appears as a sequence of characters in double quotes in the C stored function call. For example, the following is a call to a C stored function named `builtin_string_example` which takes two string arguments ("ALEX" and "Close"):

```
builtin_string_example("ALEX", "Close")
```

A C stored function "registers" a string input argument in the same way as any other argument. Namely, `builtin_string_example` may "register" to receive two string arguments (relation name and column name) with the following code in its `cfunc_instance_init_defaults` functions as follows:

```
int num = 2;
//string args
xmim_cfunc_string_args *str = new xmim_cfunc_string_args;
str->size = num;
str->names = new char*[num]
str->names[0] = "relation";
str->names[1] = "column";
str->ptr = new char*[num];
for (int i = 0; i < num; i++)
    str->ptr[i] = NULL;

args->string_args = str;
```

String input arguments are handled in the same way as the rest of the arguments.

The `cfunc_instance_get_characteristic_time_series` function is optional, but when defined, is expected to set the `args->characteristic_time_series` member to the appropriate characteristic time series, such that the time series has a non-NaN value when the date exists and a NaN value for the times when the date does not exist. For example, if a stored function is defined at exactly the times when the Close of IBM is defined, then `args->characteristic_time_series` should be set to the time series retrieved via the `xmim_cfunc_get_time_series` function for the Close of IBM.

Stored Function Framework

Two kinds of C stored functions can be introduced into the MIM server: regular C stored functions and built-in C stored functions.

Regular Stored Functions

Regular C stored functions are introduced as follows. Each stored function needs to be built into a separate shared library (e.g., a .so or .dll file) named with the name of the function it contains. For example, slice.so will contain the slice C stored function. The .so file then goes into the directory <libraryDir>/attr/plugin, which is the directory that the MIM server will check for stored functions to load upon startup.

The utility functions available for the stored functions to use are in the include/xmim_cfunc_api.h file. Utility functions mostly operate on internal MIM objects hidden from the stored function implementation. Their implementation is found in the files cfunc/xmim_cfunc_api.c and plugin/xmim_cfunc_api_model.c. The second file contains the functions which are used by the model classes of the framework and is included into the first one. This file (cfunc/xmim_cfunc_api.c) is linked into the client, while plugin/xmim_cfunc\api.c is linked into the server.

The current list of helper functions is provided below.

```
//create new xmim_cfunc_args struct  xmim_cfunc_args *xmim_cfunc_new_cfunc_args ();

//set the error string of args to message
void xmim_cfunc_set_error (xmim_cfunc_args *args, char *message);

//free error string
void xmim_cfunc_free_error (xmim_cfunc_args *args);

//creates a xmim_cfunc_date_time struct initialized to the values provided
xmim_cfunc_date_time *xmim_cfunc_get_date_time
    (xmim_cfunc_args *args,
     unsigned year, unsigned month, unsigned day,
     unsigned hour, unsigned minute,
     unsigned second);

//creates a xmim_cfunc_date_time struct initialized to the values provided
xmim_cfunc_date_time *xmim_cfunc_get_full_date_time
    (xmim_cfunc_args *args,
     unsigned year, unsigned month,
     unsigned day, unsigned_week_day,
     unsigned day_of_month, unsigned hour,

     unsigned minute, unsigned second);
//creates a xmim_cfunc_date_time struct initialized to

//the values provided, values include milliseconds
xmim_cfunc_date_time *xmim_cfunc_get_full_date_time_with_millis
    (xmim_cfunc_args *args,
     unsigned year, unsigned month,
     unsigned day,
     unsigned week_day,
     unsigned day_of_month,
     unsigned hour, unsigned minute,
```

```

        unsigned second,
        unsigned millisecond);
//creates a xmim_cfunc_date_time struct initialized to
//the values provided, values include milliseconds
xmim_cfunc_date_time *xmim_cfunc_get_date_time_with_millis
(xmim_cfunc_args *args,
 unsigned year, unsigned month,
 unsigned day,
 unsigned hour, unsigned minute,
 unsigned second, unsigned millisecond);

void xmim_cfunc_set_date (xmim_cfunc_args *args,
 xmim_cfunc_date_time *dt,
 unsigned year, unsigned month, unsigned day);

//sets the fields of dt to the specified values
void xmim_cfunc_set_full_date_time
(xmim_cfunc_args *args,
 xmim_cfunc_date_time *dt,
 unsigned year, unsigned month, unsigned day,
 unsigned week_day, unsigned day_of_month,

 unsigned hour, unsigned minute, unsigned second);

//sets the fields of dt to the specified values, values include milliseconds
void xmim_cfunc_set_full_date_time_with_millis
(xmim_cfunc_args *args,
 xmim_cfunc_date_time *dt,
 unsigned year, unsigned month, unsigned day,
 unsigned week_day, unsigned day_of_month,
 unsigned hour, unsigned minute,
 unsigned second, unsigned millisecond);

//returns a MimDateTime object initialized to the values in dt
void *xmim_cfunc_get_mim_date_time
(xmim_cfunc_args *args,
 xmim_cfunc_date_time *dt);

//get the value of ExObject pointed to by ex_object on date time dt
double xmim_cfunc_get_value
(xmim_cfunc_args *args,
 void *ex_object, xmim_cfunc_date_time *dt);

//compares the arguments passed to a cfunc
//returns 1 if they are equal, 0 otherwise
int xmim_cfunc_equal (xmim_cfunc_args *arg1, xmim_cfunc_args *arg2);

//Applies time offset named time_offset_name to date time date
//and returns a pointer to the new date time struct.
//Here time_series_name and time_offset_name are the names of
//the time series and time offset input parameters correspondingly.
//So, the name of the time series might be args->attr_args->names[0]
//and the name of the time offset may be args->offset_args->names[0].
//Delegates the functionality to
//xmim_cfunc_date_time *xmim_cfunc_apply_time_offset
//
//      (xmim_cfunc_args *args,
//      xmim_cfunc_date_time *date,
//      void *time_series,
//      void *time_offset)
xmim_cfunc_date_time *xmim_cfunc_apply_time_offset_to_args
(xmim_cfunc_args *args,
 xmim_cfunc_date_time *date,
 char *time_series_name,

```

```

        char *time_offset_name);

//Applies time offset time_offset to date time date
//and returns a pointer to the new date time struct.
//Uses an ExObject called time_series to perform the offset.
//Returns NULL and sets the error message on error.
xmim_cfunc_date_time *xmim_cfunc_apply_time_offset
    (xmim_cfunc_args *args,
     xmim_cfunc_date_time *date,
     void *time_series,
     void *time_offset);

//Applies time period called period_name to the date time dt.
//Uses ExObject pointed to by an attr arg called time_series_name
//to apply the period.
//Returns 1 upon success, 0 on failure (and sets the error message).
int xmim_cfunc_apply_time_period
    (xmim_cfunc_args *args,
     xmim_cfunc_date_time *dt, char *period_name,
     char *time_series_name,
     int include_left, int include_right,
     xmim_cfunc_date_time *left_date,
     xmim_cfunc_date_time *right_date);

//Populates the list of relations based on path.
xmim_cfunc_relation_list* xmim_cfunc_get_relations_by_name
    (xmim_cfunc_args *args,
     char *path);

//Populates the list of slice-specific relations based on path for slice C Stored Functions.
xmim_cfunc_date_relation_list* xmim_cfunc_get_date_relations_by_name
    (xmim_cfunc_args *args,
     char *path);

//Returns a pointer to an empty time series.
void *xmim_cfunc_get_empty_time_series (xmim_cfunc_args *args, void *ex_object);

//Returns a pointer to the ExRelcol Object based on the given relname and colname.
void *xmim_cfunc_get_time_series (xmim_cfunc_args *args, char *relname, char *colname);

//Sets int value for the time series object (TsTimeSeries is assumed)
//produced by on date time dt cfunc_get_empty_time_series
int xmim_cfunc_set_int_value_for_time_series
    (xmim_cfunc_args *args,
     void *time_series, int value,
     xmim_cfunc_date_time *dt);

//Sets double value for the time series object (TsTimeSeries is assumed)
//produced by on date time dt cfunc_get_empty_time_series
int xmim_cfunc_set_double_value_for_time_series
    (xmim_cfunc_args *args,
     void *time_series, double value,
     xmim_cfunc_date_time *dt);

//Gets the value of the time series (TsTimeSeries is assumed) time_series
//on date time dt.
double xmim_cfunc_get_value_for_time_series
    (xmim_cfunc_args *args,
     void *time_series, xmim_cfunc_date_time *dt);

//Sets the args->trading_pattern to the ExTradingPattern
//computed by ex_object

```

```

void xmim_cfunc_compute_trading_pattern
    (xmim_cfunc_args *args,
     void *ex_object);

//Sets the args->trading_date_range to the ExTradingPattern
//computed by ex_object
void xmim_cfunc_compute_trading_date_range
    (xmim_cfunc_args *args,
     void *ex_object);

//Returns a pointer to the ExDateRange object initialized
//to from and to.
void *xmim_cfunc_get_ex_trading_date_range
    (xmim_cfunc_args *args,
     xmim_cfunc_date_time *from,
     xmim_cfunc_date_time *to);

//Returns a pointer to the mTimeOffset object initialized
//to multiple, unit and direction.
void *xmim_cfunc_get_time_offset
    (xmim_cfunc_args *args,
     int multiple, xmim_cfunc_dwmqy unit,
     xmim_cfunc_direction direction);

//Returns a pointer to the mTimePeriod object initialized
//to fromOffset and toOffset. Defaults to today for NULL inputs.

void *xmim_cfunc_get_time_period
    (xmim_cfunc_args *args,
     void *fromOffset, void *toOffset);

//compare val to NaN
int xmim_cfunc_isNaN (xmim_cfunc_args *args,
                     double val);

//make a not a number object out of val
void xmim_cfunc_makeNan (xmim_cfunc_args *args,
                        double *val);

xmim_cfunc_date_time *xmim_cfunc_get_from_date
    (xmim_cfunc_args *args,
     void *ex_object);

xmim_cfunc_date_time *xmim_cfunc_get_to_date
    (xmim_cfunc_args *args,
     void *ex_object);

//sets an argument of type type called name to value
int xmim_cfunc_set_arg (xmim_cfunc_args *args,
                       xmim_cfunc_arg_type type, void *value, char *name);

//sets a constant arg called name to k
int xmim_cfunc_set_int_constant (xmim_cfunc_args *args, int k, char *name);

//sets a constant arg called name to k
int xmim_cfunc_set_dbl_constant (xmim_cfunc_args *args, double k, char *name);

//sets a category arg called name to cat
int xmim_cfunc_set_category (xmim_cfunc_args *args, char *cat, char *name);

//sets a string arg called name to str
void xmim_cfunc_set_string (xmim_cfunc_args *args, char *str, char *name);

```

```

//sets an attr arg called name to attr
int xmim_cfunc_set_attr (xmim_cfunc_args *args, void *attr, char *name);

//sets a time offset arg called name to offset
int xmim_cfunc_set_offset (xmim_cfunc_args *args, void *offset, char *name);

//sets a time period arg called name to period
int xmim_cfunc_set_period (xmim_cfunc_args *args, void *period, char *name);

//sets a security arg called name to relation
int xmim_cfunc_set_security (xmim_cfunc_args *args, void *relation, char *name);

//sets the xmim_cfunc state called name to state
int xmim_cfunc_set_state (xmim_cfunc_args *args, void *state, char *name);

//returns a pointer to an argument of type type called name
//returns NULL and sets the error message if an argument with
//such name is not found
void *xmim_cfunc_get_arg (xmim_cfunc_args *args, xmim_cfunc_arg_type type, char *name);

//returns a pointer to a constant called name
void *xmim_cfunc_get_int_constant (xmim_cfunc_args *args, char *name);

//returns a pointer to a constant called name
void *xmim_cfunc_get_dbl_constant (xmim_cfunc_args *args, char *name);

//returns a pointer to a category called name
void *xmim_cfunc_get_category (xmim_cfunc_args *args, char *name);

//returns a pointer to a string called name
void *xmim_cfunc_get_string (xmim_cfunc_args *args, char *name);

//returns a pointer to an attr called name
void *xmim_cfunc_get_attr (xmim_cfunc_args *args, char *name);

//returns a pointer to a time offset called name
void *xmim_cfunc_get_offset (xmim_cfunc_args *args, char *name);

//returns a pointer to a time period called name
void *xmim_cfunc_get_period (xmim_cfunc_args *args, char *name);

//returns a pointer to a security called name
void *xmim_cfunc_get_security (xmim_cfunc_args *args, char *name);

//returns a pointer to a state called name
void *xmim_cfunc_get_state (xmim_cfunc_args *args, char *name);

//frees the args struct
void xmim_cfunc_free_cfunc_args (xmim_cfunc_args *args);

//free date time dt
void xmim_cfunc_free_date_time (xmim_cfunc_args *args, xmim_cfunc_date_time *dt);

//free MimDateTime object
void xmim_cfunc_free_mim_date_time (xmim_cfunc_args *args, void *dt);

//free the time series created by the xmim_cfunc_get_time_series
//function. Cfuncs MUST use this function to destroy the time
//series.
void xmim_cfunc_free_time_series (xmim_cfunc_args *args, void *time_series);

// free the time series created via cfunc_get_empty_time_series

```

```

// function. ex_object is a pointer to the ExObject instance
// used as an input parameter in xmim_cfunc_get_empty_time_series
void xmim_cfunc_free_ts_time_series (xmim_cfunc_args *args, void *time_series);

//free trading pattern created by cfunc_compute_trading_pattern
//function
void xmim_cfunc_free_trading_pattern (xmim_cfunc_args *args;

//free trading date range created by xmim_cfunc_get_ex_trading_date_range
//function
void xmim_cfunc_free_trading_date_range (xmim_cfunc_args *args);

//free time offset created by xmim_cfunc_get_time_offset
void xmim_cfunc_free_time_offset (xmim_cfunc_args *args, void *offset);

//free time period created by xmim_cfunc_get_time_period
void xmim_cfunc_free_time_period (xmim_cfunc_args *args, void *period);

//print part of the args
void xmim_cfunc_print (xmim_cfunc_args *args);

//print time offset
void xmim_cfunc_print_time_offset (xmim_cfunc_args *args, void *off);

//print time period
void xmim_cfunc_print_time_period (xmim_cfunc_args *args, void *period);

```

Built-in Stored Functions

Built-in C stored functions are literally “built” into the MIM server, that is the `BUILTIN_CSTORED_FUNC_SRCS` target of the Makefile needs to be edited to include a particular built-in C stored function source code (to be compiled and later linked into the MIM server).

Built-in C stored functions subclass the `model/mBuiltinCStoredFunction` class, which itself is abstract. All methods (except `get_path`) that `mBuiltinCStoredFunction` subclasses are required to implement are in one-to-one correspondence with the regular plugin functions as is obvious from the names. As with regular C stored functions, `compute_trading_pattern` and `compute_trading_date_range` methods are optional. The `get_path` method is specific to built-in C stored functions and returns the path (within macro's hierarchy) and the name of the given built-in C stored function. For example, the following code in the constructor of `CStoredFuncStringExample` initializes its path:

```

this->path = new char[1028];
sprintf (this->path, "%s%s%s", ATTR_MACROS, PATH_SEPARATOR, "builtin_string_example");

```

Please note that `"builtin_string_example"` corresponds to the built-in C stored function's name, so the path to this c stored function is `"attr/builtin_string_example"`.

Implementation for the other methods of `mBuiltinCStoredFunction` subclasses is similar to regular C stored functions. Built-in C stored function sources can be placed anywhere, but it is our suggestion to put them under `plugin/` directory. We also suggest to prefix built-in C stored function source file names with `cstored_func_` (as in `cstored_func_string_example.C`) and to prefix the class names with `CStoredFunc` (as in `CStoredFuncStringExample`).

There are two examples of built-in C stored functions available. The first one is available under `plugin/cstored_func_lazy_predictor.C` and mirrors the regular C stored function under `apps/plugins/lazy_predictor.c`. The second one is under `plugin/cstored_func_string_example.C` (we will refer to it as string example function). The string example function demonstrates the use of string as input parameters and takes two input string arguments. The input arguments are user-supplied relation name and column name. The function then returns the value corresponding to the given relation and column.

Built-in C stored functions should be "registered" via the `register_builtin_cstored_func` method of the `xmim_server` right after the creation of the `xmim_server` instance in `main/api_svr_main.C`. The registration of the two example built-in C stored functions is commented out in `api_svr_main.C`. Just uncomment those lines to query the example built-in C stored functions.

Stored Function Life Cycle

The MIM server will call stored function's `cfunc_instance_init_defaults` function when the stored function is loaded. Stored function's `cfunc_instance_init` function is called when the executable stored function object is created (in the constructor of `ExCfuncObject`). The optional `cfunc_instance_get_characteristic_time_series` is called when setting the query's parameters to take the stored function's characteristic series into account. The optional `cfunc_instance_compute_trading_pattern` and `cfunc_instance_compute_trading_date_range` functions are called to ask a stored function for its trading pattern and trading date range if any. The server calls the `cfunc_instance_get_value` function for different date times to get the value of a stored function for that date and time. And finally, the `cfunc_instance_destroy` function is called upon deletion of the executable stored function object.

Example: Providing a C Stored Function Wrapper for an Existing C Function

Straightforward Approach

Suppose we have a function `compute_predictor` that takes in two arrays of values, `x` and `y`, and uses these to predict the next value of `x`. The actual implementation of this model is irrelevant to this discussion; we assume that such a model already exists. The model can be called as follows:

```
typedef struct {
    //date
    unsigned day;
    unsigned month;
    unsigned year;
    unsigned week_day;
    unsigned day_of_month;
} date;

double *compute_predictor (date *dates, double *x_values, double *y_values,
                           int size) {
    ...;
}
```

The function takes an array of dates and two arrays of values corresponding to the dates, all of which are of length `size`. Here, `x_values[0]` is the value of a time series `X` on the date `dates[0]`.

Now let us demonstrate the straightforward way of converting this function into a C stored function that the framework can work with. This method does not require any modifications to the function that implements the model. In the next section, we will describe a more efficient way to do this, which does involve rewriting the model.

Let `predictor` be the name for the C stored function we want to write. The first thing we need to do is specify the input parameters for the C stored function, which in this case are going to be two security parameters. Therefore, we can define the `cfunc_instance_init_defaults` function as follows:

```
#define PRED_STOCK1 "stock1"
#define PRED_STOCK2 "stock2"

extern "C" int cfunc_instance_init_defaults (xmim_cfunc_args *args) {
    int num = 2;
    //security args
    xmim_cfunc_security_args *sec = new xmim_cfunc_security_args;
    sec->size = num;
    sec->names = new char*[num];
    sec->names[0] = PRED_STOCK1;
    sec->names[1] = PRED_STOCK2;
    sec->rels = new char*[num];
    sec->rels[0] = NULL;
    sec->rels[1] = NULL;

    args->security_args = sec;

    //define the state here
```

```

    return 1;
}

```

A state is specific to a particular C stored function and is defined in the `cfunc_instance_init_defaults` function. A state consists of all objects that need to persist between invocations of different functions, so that a function can retrieve an object it is interested in and use for its purposes.

Now let us define the state which our C stored function needs to keep. We need to store one of the time series that we will create based on the input stock symbols so that we are able to define the trading pattern and the trading range for `predictor` later. Also, in `cfunc_instance_init` function we will compute the result values and "remember" them throughout our C stored function's lifetime.

Taking this into account the `cfunc_instance_init_defaults` looks as follows:

```

#define PRED_STOCK1 "stock1"
#define PRED_STOCK2 "stock2"
#define PRED_TIME_SERIES "time_series"
#define PRED_RESULT "result"

#define DEFAULT_COLUMN "Asks"

extern "C" int cfunc_instance_init_defaults (xmim_cfunc_args *args) {
    int num = 2;
    //security args
    xmim_cfunc_security_args *sec = new xmim_cfunc_security_args;
    sec->size = num;
    sec->names = new char*[num];
    sec->names[0] = PRED_STOCK1;
    sec->names[1] = PRED_STOCK2;
    sec->rels = new char*[num];
    sec->rels[0] = NULL;
    sec->rels[1] = NULL;

    args->security_args = sec;

    xmim_cfunc_state *state = new xmim_cfunc_state;
    num = 2;
    state->size = num;
    state->ptr = new xmim_cfunc_obj_ptr[num];
    for (int i = 0; i < num; i++)
        state->ptr[i].obj = NULL;

    state->names = new char*[num];
    state->names[0] = PRED_TIME_SERIES;
    state->names[1] = PRED_RESULT;

    args->state = state;

    return 1;
}

```

In the `cfunc_instance_init` function C stored functions are able to access the input parameters with which they are called from a user's query. For example, if our `predictor` is invoked via the MIM query:

```

SHOW
    predictor(EA.EPWWQ304, EA.EPWWQ404)
WHEN
    Date is after 2002
AND
    Date is Wednesday

```

Then, to get the first input stock symbol, we would use the `xmim_cfunc_get_arg` function:

```
char *symbol = NULL;
symbol = (char *)xmim_cfunc_get_arg (args, P_SECURITY, PRED_STOCK1);
if (!symbol)
    return 0;
```

The first argument to `xmim_cfunc_get_arg` is an `xmim_cfunc_args` structure, which is populated with the input parameters after the call to `cfunc_instance_init_defaults`. This means that the input parameters are only inaccessible in the `cfunc_instance_init_defaults` function and accessible in all other functions of a C stored function. In general, we suggest that the `cfunc_instance_init_defaults` should only be used to specify the input parameters for the C stored function and its state. Nothing else should happen there since the `xmim_cfunc_args` structure is empty at that point.

The second argument is the type of the input parameter we want to get back. All possible types are defined in:

```
//arg types for a cfunc
typedef enum {
    P_INT_CONSTANT,
    P_DBL_CONSTANT,
    P_CATEGORY,
    P_STRING,
    P_ATTR,
    P_TIME_OFFSET,
    P_TIME_PERIOD,
    P_SECURITY,
    //cfunc specific state
    P_STATE
} xmim_cfunc_arg_type;
```

in the `xmim_cfunc_api.h` file. And the third argument to `xmim_cfunc_get_arg` is the name of the input parameter as we defined it in the `cfunc_instance_init_defaults` function.

The last two lines of the code snippet above handle an exceptional condition when the returned symbol is NULL. `xmim_cfunc_get_arg` will return NULL in two cases: when it fails to retrieve the named parameter or if the parameter value is NULL. In the first case `args->error` is set to indicate the failure, while in the second case no error is set. So, if a NULL can be a valid value for a parameter (the state is treated analogously to input parameters by `xmim_cfunc_get_arg`, and NULL can be a valid value before we initialize the state), then we need to check whether `args->error` is set (in which case `xmim_cfunc_get_arg` has failed) or is NULL (in which case the parameter value is NULL).

Returning a 0 from any of the C stored function's functions signals that an error has occurred during the function's execution. The framework will then raise an error with the `args->error` error message.

Now let us get back to what we want to do in the `cfunc_instance_init` function. Given the input stock symbols, we are going to create the corresponding time series. The `compute_predictor` function takes three arrays as its input: an array of dates and arrays of values on those dates for the two time series. This is what we could store in our state. However, it is more convenient for us to precompute the result exactly the way we compute it in the `compute_predictor` function and then store it in our state. Thus the `pred_result` structure:

```
typedef struct {
    double *values;
    xmim_cfunc_date_time **dates;
    int size;
} pred_result;
```

which stores both the dates array and the result values on the corresponding dates.

In general, the purpose of the `cfunc_instance_init` function is to initialize the state. It is executed before the first call to the `cfunc_instance_get_value`, so all the initialization should be done there.

Our `cfunc_instance_init` will initialize the state as follows:

```
extern "C" int cfunc_instance_init (xmim_cfunc_args *args) {
    char *relname1 = NULL, *relname2 = NULL;
    void *series1 = NULL, *series2 = NULL;
    xmim_cfunc_date_time *from_date = NULL, *to_date = NULL, *date = NULL, *old_date = NULL;
    void *offset;
    pred_result *result = NULL;
    int size = 0;

    //first input argument
    relname1 = (char *)xmim_cfunc_get_arg (args, P_SECURITY, PRED_STOCK1);
    if (!relname1)
        return 0;

    //second input argument
    relname2 = (char *)xmim_cfunc_get_arg (args, P_SECURITY, PRED_STOCK2);
    if (!relname2)
        return 0;

    //create the time series for the first stock symbol
    series1 = xmim_cfunc_get_time_series (args, relname1, DEFAULT_COLUMN);
    if (!series1)
        return 0;
    //store the first time series in the state
    if (!xmim_cfunc_set_arg (args, P_STATE, series1, PRED_TIME_SERIES))
        return 0;

    //create the time series for the second stock symbol
    //will get deleted by the framework
    series2 = xmim_cfunc_get_time_series (args, relname2, DEFAULT_COLUMN);
    if (!series2)
        return 0;

    // the from and to dates for the first time series
    // the date range we are going to use
    from_date = xmim_cfunc_get_from_date (args, series1);
    to_date = xmim_cfunc_get_to_date (args, series1);

    //calculate the number of values we have
    date = xmim_cfunc_get_from_date (args, series1);
    size = 1;

    while (!xmim_cfunc_date_time_equal (args, date, to_date)) {
        offset = xmim_cfunc_get_time_offset (args, 1, PL_DAYS, PL_LATER);
        old_date = date;
        date = xmim_cfunc_apply_time_offset (args, date, series1, offset);
        xmim_cfunc_free_date_time (args, old_date);
        xmim_cfunc_free_time_offset (args, offset);
        size++;
    }

    //compute the result
    result = new pred_result;
    result->values = new double[size];
    result->dates = new xmim_cfunc_date_time*[size];
    result->size = size;
}
```

```

double v1, v2, prev_v1, prev_v2;
//the first value is a NaN since we do not have the
//value on the previous date
double zero = 0;
double *doubleNan = &zero;
xmim_cfunc_makeNan (args, doubleNan);
result->values[0] = *doubleNan;
result->dates[0] = from_date;
prev_v1 = xmim_cfunc_get_value (args, series1, from_date);
prev_v2 = xmim_cfunc_get_value (args, series2, from_date);
//compute the values using the two time series
for (int i = 1; i < size; i++) {
    offset = xmim_cfunc_get_time_offset (args, i, PL_DAYS, PL_LATER);
    date = xmim_cfunc_apply_time_offset (args, from_date, series1, offset);
    v1 = xmim_cfunc_get_value (args, series1, date);
    v2 = xmim_cfunc_get_value (args, series2, date);
    result->values[i] = v1 +
        (0.5 * v1 *
         ((v1 - prev_v1) / prev_v1) +
         ((v2 - prev_v2) / prev_v2));

    result->dates[i] = date;
    prev_v1 = v1;
    prev_v2 = v2;
    xmim_cfunc_free_time_offset (args, offset);
}

//store the result in the state
if (!xmim_cfunc_set_arg (args, P_STATE, result, PRED_RESULT))
    return 0;

return 1;
}

```

`xmim_cfunc_get_time_series` returns a time series object corresponding to the relation and the column provided. The column is assumed to be the same for all symbols (relations) and known beforehand. A time series can be either destroyed explicitly by calling the

```
void xmim_cfunc_free_time_series (xmim_cfunc_args *args, void *time_series);
```

function or destroyed implicitly by the framework when the C stored function instance is being destroyed (note, that this only holds for time series objects created by the framework, and everything you create, you should destroy).

When computing the values for the result array, we are using the `xmim_cfunc_get_time_offset` function to get the desired offset and then get a date by applying that offset to the `from_date`.

Then, the `cfunc_instance_get_value` function will retrieve the result we have computed in the `cfunc_instance_init` function, find a value on the requested date and return that value. It looks as follows:

```

extern "C" int cfunc_instance_get_value (xmim_cfunc_args *args, xmim_cfunc_date_time *dt) {
    pred_result *result = (pred_result *)xmim_cfunc_get_arg (args, P_STATE, PRED_RESULT);
    xmim_cfunc_date_time *date = NULL;

    if (!result)
        return 0;

    for (int i = 0; i < result->size; i++) {

```

```

        date = (xmim_cfunc_date_time *) result->dates[i];
        if (xmim_cfunc_date_time_equal (args, date, dt)) {
            args->result = result->values[i];
            return 1;
        }
    }
    //we do not have the requested date - return NaN.
    double zero = 0;
    double *doubleNan = &zero;
    xmim_cfunc_makeNan (args, doubleNan);
    args->result = *doubleNan;

    return 1;
}

```

As you have noticed, the `cfunc_instance_get_value` function returns the result by setting `args->result` to the value it needs to return.

Then, finally in the `cfunc_instance_destroy` function we are going to clean up everything we have created:

```

extern "C" int cfunc_instance_destroy (xmim_cfunc_args *args) {
    delete[] args->security_args->rels;
    delete[] args->security_args->names;
    delete args->security_args;
    args->security_args = NULL;

    //the time series will be deleted for us by the framework

    pred_result *result = (pred_result *)xmim_cfunc_get_arg (args, P_STATE, PRED_RESULT);
    if (result) {
        for (int i = 0; i < result->size; i++)
            delete result->dates[i];

        delete[] result->dates;
        delete[] result->values;

        delete result;
    }

    return 1;
}

```

We covered the mandatory functions each C stored function is *required* to implement. The next two functions are optional: `cfunc_instance_compute_trading_pattern` and `cfunc_instance_compute_trading_date_range`.

They define the trading pattern (e.g, M-F, 8-5) and the trading date range (e.g, Jan 1, 1923 -- November 15, 2004) respectively and can be implemented by using one of the existing time series to get the trading pattern and the trading date range:

```

extern "C" int cfunc_instance_compute_trading_pattern (xmim_cfunc_args *args) {
    void *time_series = xmim_cfunc_get_arg (args, P_STATE, STOCK1_TIME_SERIES);

    if (!time_series)
        return 0;

    xmim_cfunc_compute_trading_pattern (args, time_series);

    return 1;
}

```

```

extern "C" int cfunc_instance_compute_trading_date_range (xmim_cfunc_args *args) {
    void *time_series = xmim_cfunc_get_arg (args, P_STATE, STOCK1_TIME_SERIES);

    if (!time_series)
        return 0;

    xmim_cfunc_compute_trading_date_range (args, time_series);

    return 1;
}

extern "C" int cfunc_instance_get_characteristic_time_series (xmim_cfunc_args *args) {
    void *close_series = xmim_cfunc_get_arg (args, P_STATE, VWAP_TEST_CLOSE_SERIES);
    if (!close_series)
        close_series = vwap_set_series (args);

    if (!close_series) {
        xmim_cfunc_set_error (args, "VWAP get_characteristic_time_series: failed to create the time
series. Please, make sure this call is happening at the right place.");
        return 0;
    }

    args->characteristic_time_series = close_series;

    return 1;
}

```

Complete code for this example is available in [“Complete Predictor Code”](#).

Lazy Evaluation Approach

Now let us consider a more efficient way of writing a C stored function based on the [example](#) we described in the previous section. The code in this section is more efficient in two ways:

- It does not have to allocate a time series to store all possible return values and
- It does not compute a result unless it is actually needed by the query.

However, to achieve this efficiency we must modify the code that computes the predictor model. It should be noted that this modification may be impractical or impossible. Let us call the new C stored function we are going to write `lazy_predictor`.

The only code that would change in the `cfunc_instance_init_defaults` function is the code that handles the state, since now we want to store the time series corresponding to the input stock symbols in the state:

```

xmim_cfunc_state *state = new xmim_cfunc_state;
num = 2;
state->size = num;
state->ptr = new xmim_cfunc_obj_ptr[num];
for (int i = 0; i < num; i++)
    state->ptr[i].obj = NULL;

state->names = new char*[num];
state->names[0] = STOCK1_TIME_SERIES;
state->names[1] = STOCK2_TIME_SERIES;

```

In the `cfunc_instance_init` function we are going to create the time series corresponding to the two stock input symbols and store them in the state (just like we did in the previous section).

The `cfunc_instance_get_value` is going to compute the result value for the specific date it is asked about.

```
extern "C" int cfunc_instance_get_value (xmim_cfunc_args *args, xmim_cfunc_date_time *dt) {
    xmim_cfunc_date_time *from_date = NULL;
    void *series1 = xmim_cfunc_get_arg (args, P_STATE, STOCK1_TIME_SERIES);
    void *series2 = xmim_cfunc_get_arg (args, P_STATE, STOCK2_TIME_SERIES);

    if (!series1 || !series2)
        return 0;

    //return a NaN for the first date
    from_date = xmim_cfunc_get_from_date (args, series1);
    if (xmim_cfunc_date_time_equal (args, from_date, dt)) {
        double zero = 0;
        double *doubleNan = &zero;
        xmim_cfunc_makeNan (args, doubleNan);
        args->result = *doubleNan;
        return 1;
    }

    //compute the result
    double v1, v2, prev_v1, prev_v2;
    xmim_cfunc_date_time *date = NULL;
    void *offset = NULL;
    offset = xmim_cfunc_get_time_offset (args, 1, PL_DAYS, PL_AGO);
    date = xmim_cfunc_apply_time_offset (args, dt, series1, offset);
    prev_v1 = xmim_cfunc_get_value (args, series1, date);
    prev_v2 = xmim_cfunc_get_value (args, series2, date);
    v1 = xmim_cfunc_get_value (args, series1, dt);
    v2 = xmim_cfunc_get_value (args, series2, dt);
    xmim_cfunc_free_date_time (args, date);
    xmim_cfunc_free_time_offset (args, offset);

    args->result = v1 +
        (0.5 * v1 *
            ((v1 - prev_v1) / prev_v1) +
            ((v2 - prev_v2) / prev_v2));

    return 1;
}
```

The rest of the functions here are analogous to the previous section.

The advantage of this approach is computing the values "on demand" which is faster than pre-computing and storing them.

When should either approach be used? We suggest that if you are just sitting down to write a new C stored function you try to use the lazy evaluation approach. However, you are more likely to have some functions you would like to "wrap around" to use within the framework. In this case, if the logic of a particular function you want to "wrap around" is easy to change such that it uses lazy evaluation, we would recommend using the lazy evaluation approach.

If the function is complicated, then it might be better to "wrap it around" in a straightforward way.

Complete code for this example is available in ["Complete Lazy Predictor Code"](#).

Complete Predictor Code

```

#include "include/xmim_cfunc_api.h"
#include <stdio.h>

//names for the input parameters
#define PRED_STOCK1 "stock1"
#define PRED_STOCK2 "stock2"
//names for the state parameters
#define PRED_TIME_SERIES "time_series"
#define PRED_RESULT "result"

//column to use when creating a time
//series for the stock symbols
#define DEFAULT_COLUMN "Asks"

//struct to store in the state
typedef struct {
    double *values;
    xmim_cfunc_date_time **dates;
    int size;
} pred_result;

//specify the input parameters and the state
extern "C" int cfunc_instance_init_defaults (xmim_cfunc_args *args) {
    int num = 2;
    //security args
    xmim_cfunc_security_args *sec = new xmim_cfunc_security_args;
    sec->size = num;
    sec->names = new char*[num];
    sec->names[0] = PRED_STOCK1;
    sec->names[1] = PRED_STOCK2;
    sec->rels = new char*[num];
    sec->rels[0] = NULL;
    sec->rels[1] = NULL;

    args->security_args = sec;

    xmim_cfunc_state *state = new xmim_cfunc_state;
    num = 2;
    state->size = num;
    state->ptr = new xmim_cfunc_obj_ptr[num];
    for (int i = 0; i < num; i++)
        state->ptr[i].obj = NULL;

    state->names = new char*[num];
    state->names[0] = PRED_TIME_SERIES;
    state->names[1] = PRED_RESULT;

    args->state = state;

    return 1;
}

//initialize the state
extern "C" int cfunc_instance_init (xmim_cfunc_args *args) {
    char *relname1 = NULL, *relname2 = NULL;
    void *series1 = NULL, *series2 = NULL;
    xmim_cfunc_date_time *from_date = NULL, *to_date = NULL, *date = NULL, *old_date = NULL;
    void *offset;
    pred_result *result = NULL;

```

```

int size = 0;

//first input argument
relname1 = (char *)xmim_cfunc_get_arg (args, P_SECURITY, PRED_STOCK1);
if (!relname1)
    return 0;

//second input argument
relname2 = (char *)xmim_cfunc_get_arg (args, P_SECURITY, PRED_STOCK2);
if (!relname2)
    return 0;

//create the time series for the first stock symbol
series1 = xmim_cfunc_get_time_series (args, relname1, DEFAULT_COLUMN);
if (!series1)
    return 0;
//store the first time series in the state
if (!xmim_cfunc_set_arg (args, P_STATE, series1, PRED_TIME_SERIES))
    return 0;

//create the time series for the second stock symbol
//will get deleted in the ~CoCompiler () [CoCompiler::deleteExObjects]
series2 = xmim_cfunc_get_time_series (args, relname2, DEFAULT_COLUMN);
if (!series2)
    return 0;

// the from and to dates for the first time series
// the date range we are going to use
from_date = xmim_cfunc_get_from_date (args, series1);
to_date = xmim_cfunc_get_to_date (args, series1);

//calculate the number of values we have
date = xmim_cfunc_get_from_date (args, series1);
size = 1;

while (!xmim_cfunc_date_time_equal (args, date, to_date)) {
    offset = xmim_cfunc_get_time_offset (args, 1, PL_DAYS, PL_LATER);
    old_date = date;
    date = xmim_cfunc_apply_time_offset (args, date, series1, offset);
    xmim_cfunc_free_date_time (args, old_date);
    xmim_cfunc_free_time_offset (args, offset);
    size++;
}

//compute the result
result = new pred_result;
result->values = new double[size];
result->dates = new xmim_cfunc_date_time*[size];
result->size = size;

double v1, v2, prev_v1, prev_v2;
//the first value is a NaN since we do not have the
//value on the previous date
double zero = 0;
double *doubleNan = &zero;
xmim_cfunc_makeNan (args, doubleNan);
result->values[0] = *doubleNan;
result->dates[0] = from_date;
prev_v1 = xmim_cfunc_get_value (args, series1, from_date);
prev_v2 = xmim_cfunc_get_value (args, series2, from_date);
//compute the values using the two time series
for (int i = 1; i < size; i++) {
    offset = xmim_cfunc_get_time_offset (args, i, PL_DAYS, PL_LATER);

```

```

    date = xmim_cfunc_apply_time_offset (args, from_date, series1, offset);
    v1 = xmim_cfunc_get_value (args, series1, date);
    v2 = xmim_cfunc_get_value (args, series2, date);
    result->values[i] = v1 +
        (0.5 * v1 *
         (((v1 - prev_v1) / prev_v1) +
          ((v2 - prev_v2) / prev_v2)));

    result->dates[i] = date;
    prev_v1 = v1;
    prev_v2 = v2;
    xmim_cfunc_free_time_offset (args, offset);
}

//store the result in the state
if (!xmim_cfunc_set_arg (args, P_STATE, result, PRED_RESULT))
    return 0;

return 1;
}

extern "C" int cfunc_instance_compute_trading_pattern (xmim_cfunc_args *args) {
    void *time_series = xmim_cfunc_get_arg (args, P_STATE, PRED_TIME_SERIES);

    if (!time_series)
        return 0;

    xmim_cfunc_compute_trading_pattern (args, time_series);

    return 1;
}

extern "C" int cfunc_instance_compute_trading_date_range (xmim_cfunc_args *args) {
    void *time_series = xmim_cfunc_get_arg (args, P_STATE, PRED_TIME_SERIES);

    if (!time_series)
        return 0;

    xmim_cfunc_compute_trading_date_range (args, time_series);

    return 1;
}

extern "C" int cfunc_instance_get_value (xmim_cfunc_args *args, xmim_cfunc_date_time *dt) {
    pred_result *result = (pred_result *)xmim_cfunc_get_arg (args, P_STATE, PRED_RESULT);
    xmim_cfunc_date_time *date = NULL;

    if (!result)
        return 0;

    for (int i = 0; i < result->size; i++) {
        date = (xmim_cfunc_date_time *) result->dates[i];
        if (xmim_cfunc_date_time_equal (args, date, dt)) {
            args->result = result->values[i];
            return 1;
        }
    }
}

//we do not have the requested date - return NaN.
double zero = 0;
double *doubleNan = &zero;
xmim_cfunc_makeNan (args, doubleNan);
args->result = *doubleNan;

```

```
    return 1;
}

extern "C" int cfunc_instance_destroy (xmim_cfunc_args *args) {
    delete[] args->security_args->rels;
    delete[] args->security_args->names;
    delete args->security_args;
    args->security_args = NULL;

    //the RelCol object pointed to by args->state->ptr[0].obj
    //has already been deleted for us
    //in the ~CoCompiler () [CoCompiler::deleteExObjects]

    pred_result *result = (pred_result *)xmim_cfunc_get_arg (args, P_STATE, PRED_RESULT);
    if (result) {
        for (int i = 0; i < result->size; i++)
            delete result->dates[i];

        delete[] result->dates;
        delete[] result->values;

        delete result;
    }

    return 1;
}
```

Complete Lazy Predictor Code

```

#include "include/xmim_cfunc_api.h"
#include <stdio.h>

#define PRED_STOCK1 "stock1"
#define PRED_STOCK2 "stock2"
#define STOCK1_TIME_SERIES "stock1_series"
#define STOCK2_TIME_SERIES "stock2_series"

#define DEFAULT_COLUMN "Asks"

extern "C" int cfunc_instance_init_defaults (xmim_cfunc_args *args) {
    int num = 2;
    //security args
    xmim_cfunc_security_args *sec = new xmim_cfunc_security_args;
    sec->size = num;
    sec->names = new char*[num];
    sec->names[0] = PRED_STOCK1;
    sec->names[1] = PRED_STOCK2;
    sec->rels = new char*[num];
    sec->rels[0] = NULL;
    sec->rels[1] = NULL;

    args->security_args = sec;

    xmim_cfunc_state *state = new xmim_cfunc_state;
    num = 2;
    state->size = num;
    state->ptr = new xmim_cfunc_obj_ptr[num];
    for (int i = 0; i < num; i++)
        state->ptr[i].obj = NULL;

    state->names = new char*[num];
    state->names[0] = STOCK1_TIME_SERIES;
    state->names[1] = STOCK2_TIME_SERIES;

    args->state = state;

    return 1;
}

extern "C" int cfunc_instance_init (xmim_cfunc_args *args) {
    char *relnamel = NULL, *relname2 = NULL;
    void *series1 = NULL, *series2 = NULL;

    relname1 = (char *)xmim_cfunc_get_arg (args, P_SECURITY, PRED_STOCK1);
    if (!relnamel)
        return 0;

    relname2 = (char *)xmim_cfunc_get_arg (args, P_SECURITY, PRED_STOCK2);
    if (!relname2)
        return 0;

    series1 = xmim_cfunc_get_time_series (args, relname1, DEFAULT_COLUMN);

    if (!series1)
        return 0;
    if (!xmim_cfunc_set_arg (args, P_STATE, series1, STOCK1_TIME_SERIES))
        return 0;
}

```

```

//will get deleted in the ~CoCompiler () [CoCompiler::deleteExObjects]
series2 = xmim_cfunc_get_time_series (args, relname2, DEFAULT_COLUMN);
if (!series2)
    return 0;
if (!xmim_cfunc_set_arg (args, P_STATE, series2, STOCK2_TIME_SERIES))
    return 0;

return 1;
}

extern "C" int cfunc_instance_compute_trading_pattern (xmim_cfunc_args *args) {
    void *time_series = xmim_cfunc_get_arg (args, P_STATE, STOCK1_TIME_SERIES);

    if (!time_series)
        return 0;

    xmim_cfunc_compute_trading_pattern (args, time_series);

    return 1;
}

extern "C" int cfunc_instance_compute_trading_date_range (xmim_cfunc_args *args) {
    void *time_series = xmim_cfunc_get_arg (args, P_STATE, STOCK1_TIME_SERIES);

    if (!time_series)
        return 0;

    xmim_cfunc_compute_trading_date_range (args, time_series);

    return 1;
}

extern "C" int cfunc_instance_get_value (xmim_cfunc_args *args, xmim_cfunc_date_time *dt) {
    xmim_cfunc_date_time *from_date = NULL;
    void *series1 = xmim_cfunc_get_arg (args, P_STATE, STOCK1_TIME_SERIES);
    void *series2 = xmim_cfunc_get_arg (args, P_STATE, STOCK2_TIME_SERIES);

    if (!series1 || !series2)
        return 0;

    //return a NaN for the first date
    from_date = xmim_cfunc_get_from_date (args, series1);
    if (xmim_cfunc_date_time_equal (args, from_date, dt)) {
        double zero = 0;
        double *doubleNan = &zero;
        xmim_cfunc_makeNan (args, doubleNan);
        args->result = *doubleNan;
        return 1;
    }

    //compute the result
    double v1, v2, prev_v1, prev_v2;
    xmim_cfunc_date_time *date = NULL;
    void *offset = NULL;
    offset = xmim_cfunc_get_time_offset (args, 1, PL_DAYS, PL_AGO);
    date = xmim_cfunc_apply_time_offset (args, dt, series1, offset);
    prev_v1 = xmim_cfunc_get_value (args, series1, date);
    prev_v2 = xmim_cfunc_get_value (args, series2, date);
    v1 = xmim_cfunc_get_value (args, series1, dt);
    v2 = xmim_cfunc_get_value (args, series2, dt);
    xmim_cfunc_free_date_time (args, date);
    xmim_cfunc_free_time_offset (args, offset);
}

```

```
args->result = v1 +
    (0.5 * v1 *
        ((v1 - prev_v1) / prev_v1) +
        ((v2 - prev_v2) / prev_v2));

return 1;
}

extern "C" int cfunc_instance_destroy (xmim_cfunc_args *args) {
    delete[] args->security_args->rels;
    delete[] args->security_args->names;
    delete args->security_args;
    args->security_args = NULL;

    //the RelCol objects pointed to by args->state->ptr[0].obj
    // and args->state->ptr[1].obj
    //has already been deleted for us
    //in the ~CoCompiler () [CoCompiler::deleteExObjects]

return 1;
}
```


Index

C

- C Stored Function, 1
 - cfunc_instance_compute_trading_date_range, 3, 12
 - cfunc_instance_compute_trading_pattern, 3, 12
 - cfunc_instance_destroy, 4, 12
 - cfunc_instance_get_characteristic_time_series, 4, 4, 12
 - cfunc_instance_get_value, 4, 12
 - cfunc_instance_init, 3, 12
 - cfunc_instance_init_defaults, 3, 3, 12
 - Description, 1
 - Example, 13
 - Framework, 5
 - Built-in Stored Functions, 10
 - Regular Stored Functions, 5
 - Lazy Evaluation Approach, 19
 - Lazy Predictor Code, 25
 - Life Cycle, 12
 - Predictor Code, 21
 - Structure, 2
 - Utility, 5
- compute_trading_date_range Method, 10
- compute_trading_pattern Method, 10

D

- Destruct
 - Stored Function, 4

I

- Initialize
 - Default Values, 3
 - State, 3

L

- lazy_predictor, 19

M

- Methods

- compute_trading_date_range, 10
- compute_trading_pattern, 10
- register_builtin_cstored_func, 11

R

- register_builtin_cstored_func Method, 11

T

- Time Series
 - Compute Characteristic, 4
- Trading Date Range
 - Compute, 3
- Trading Pattern
 - Compute, 3

U

- Utility Functions, 5

