C++ API Overview

This section documents the Gurobi C++ interface. This manual begins with a quick overview of the classes exposed in the interface and the most important methods on those classes. It then continues with a comprehensive presentation of all of the available classes and methods.

If you are new to the Gurobi Optimizer, we suggest that you start with the Quick Start Guide or the Example Tour. These documents provide concrete examples of how to use the classes and methods described here.

Environments

The first step in using the Gurobi C++ interface is to create an environment object. Environments are represented using the GRBEnv class. An environment acts as the container for all data associated with a set of optimization runs. You will generally only need one environment object in your program.

Models

You can create one or more optimization models within an environment. Each model is represented as an object of class GRBModel. A model consists of a set of decision variables (objects of class GRBVar), a linear or quadratic objective function on those variables (specified using GRBModel::setObjective), and a set of constraints on these variables (objects of class GRBConstr, GRBQConstr, or GRBSOS). Each variable has an associated lower bound, upper bound, and type (continuous, binary, etc.). Each linear or quadratic constraint has an associated sense (less-than-or-equal, greater-than-or-equal, or equal), and right-hand side value.

Linear constraints are specified by building linear expressions (objects of class GRBLinExpr), and then specifying relationships between these expressions (for example, requiring that one expression be equal to another). Quadratic constraints are built in a similar fashion, but using quadratic expressions (objects of class GRBQuadExpr) instead.

An optimization model may be specified all at once, by loading the model from a file (using the appropriate GRBModel constructor), or built incrementally, by first constructing an empty object of class GRBModel and then subsequently calling GRBModel::addVar or GRBModel::addVars to add additional variables, and GRBModel::addConstr or GRBModel::addQConstr to add additional constraints. Models are dynamic entities; you can always add or remove variables or constraints.

We often refer to the *class* of an optimization model. A model with a linear objective function, linear constraints, and continuous variables is a *Linear Program (LP)*. If the objective is quadratic, the model is a *Quadratic Program (QP)*. If any of the constraints are quadratic, the model is a *Quadratically-Constrained Program (QCP)*. We'll sometimes also discuss a special case of QCP, the *Second-Order Cone Program (SOCP)*. If the model contains any integer variables, semi-continuous variables, semi-integer variables, or Special Ordered Set (SOS) constraints, the model is a *Mixed Integer Program (MIP)*. We'll also sometimes discuss special cases of MIP, including *Mixed Integer Linear Programs (MILP)*, *Mixed Integer Quadratic Programs (MIQP)*, *Mixed Integer Quadratically-Constrained Programs (MIQCP)*, and *Mixed Integer Second-Order Cone Programs (MISOCP)*. The Gurobi Optimizer handles all of these model classes.

Solving a Model

Once you have built a model, you can call GRBModel::optimize to compute a solution. By default, optimize will use the concurrent optimizer to solve LP models, the barrier algorithm to solve QP and QCP models, and the branch-and-cut algorithm to solve mixed integer models. The solution is stored in a set of *attributes* of the model. These attributes can be queried using a set of attribute query methods on the GRBModel, GRBVar, GRBConstr, and GRBQConstr classes.

The Gurobi algorithms keep careful track of the state of the model, so calls to GRBModel::optimize will only perform further optimization if relevant data has changed since the model was last optimized. If you would like to discard previously computed solution information and restart the optimization from scratch without changing the model, you can call GRBModel::reset.

After a MIP model has been solved, you can call GRBModel::fixedModel to compute the associated *fixed* model. This model is identical to the input model, except that all integer variables are fixed to their values in the MIP solution. In some applications, it is useful to compute information on this continuous version of the MIP model (e.g., dual variables, sensitivity information, etc.).

Infeasible Models

You have a few options if a model is found to be infeasible. You can try to diagnose the cause of the infeasibility, attempt to repair the infeasibility, or both. To obtain information that can be useful for diagnosing the cause of an infeasibility, call GRBModel::computeIIS to compute an Irreducible Inconsistent Subsystem (IIS). This method can be used for both continuous and MIP models, but you should be aware that the MIP version can be quite expensive. This method populates a set of IIS attributes.

To attempt to repair an infeasibility, call GRBModel::feasRelax to compute a feasibility relaxation for the model. This relaxation allows you to find a solution that minimizes the magnitude of the constraint violation.

Querying and Modifying Attributes

Most of the information associated with a Gurobi model is stored in a set of attributes. Some attributes are associated with the variables of the model, some with the constraints of the model, and some with the model itself. To give a simple example, solving an optimization model causes the `X` variable attribute to be populated. Attributes such as `X` that are computed by the Gurobi optimizer cannot be modified directly by the user, while others, such as the variable lower bound (the `LB` attribute) can.

Attributes are queried using GRBVar::get, GRBConstr::get, GRBQConstr::get, or GRBModel::get, and modified using GRBVar::set, GRBConstr::set, GRBQConstr::set, or GRBModel::set. Attributes are grouped into a set of enums by type (GRB_CharAttr, GRB_DoubleAttr, GRB_IntAttr, GRB_StringAttr). The `get()` and `set()` methods are overloaded, so the type of the attribute determines the type of the returned value. Thus, `constr.get(GRB.DoubleAttr.RHS)` returns a double, while `constr.get(GRB.CharAttr.Sense)` returns a char.

If you wish to retrieve attribute values for a set of variables or constraints, it is usually more efficient to use the array methods on the associated GRBModel object. Method GRBModel::get includes signatures that allow you to query or modify attribute values for arrays of variables or constraints.

The full list of attributes can be found in the Attributes section.

Additional Model Modification Information

Most modifications to an existing model are done through the attribute interface (e.g., changes to variable bounds, constraint right-hand sides, etc.). The main exceptions are modifications to the constraint matrix and the objective function.

The constraint matrix can be modified in a few ways. The first is to call the chgCoeffs method on a GRBModel object to change individual matrix coefficients. This method can be used to modify the value of an existing non-zero, to set an existing non-zero to zero, or to create a new non-zero. The constraint matrix is also modified when you remove a variable or constraint from the model (through the GRBModel::remove method). The non-zero values associated with the deleted constraint or variable are removed along with the constraint or variable itself.

The model objective function can also be modified in a few ways. The easiest is to build an expression that captures the objective function (a GRBLinExpr or GRBQuadExpr object), and then pass that expression to method GRBModel::setObjective. If you wish to modify the objective, you can simply call `setObjective` again with a new `GRBLinExpr` or `GRBQuadExpr` object.

For linear objective functions, an alternative to `setObjective` is to use the `Obj` variable attribute to modify individual linear objective coefficients.

If your variables have piecewise-linear objectives, you can specify them using the setPWLObj method. Call this method once for each relevant variable. The Gurobi simplex solver includes algorithmic support for convex piecewise-linear objective functions, so for continuous models you should see a substantial performance benefit from using this feature. To clear a previously specified piecewise-linear objective function, simply set the `Obj` attribute on the corresponding variable to 0.

Lazy Updates

One very important item to note about attribute and model modifications in the Gurobi optimizer is that they are performed in a *lazy* fashion, meaning that they don't actually affect the model until the next call to optimize or update on that model object. This approach provides the advantage that the model remains unchanged while you are in the process of making multiple modifications. The downside, of course, is that you have to remember to call update in order to see the effect of your changes.

If you forget to call update, your program won't crash. The most common symptom of a missing update is a `NOT_IN_MODEL` exception, which indicates that the object you are trying to reference isn't in the model yet.

If you find the need to call update inconvenient, you can adjust the behavior of lazy updates with the UpdateMode parameter. By setting this parameter to 1, you can use newly added variables and constraints immediately for building or modifying the model. This setting does have a few downsides, though. It causes Gurobi to use a small amount of additional internal storage, and it introduces a small performance overhead. In addition, this setting may cause Gurobi to make less aggressive use of warm-start information when you modify a model and resolve it using simplex.

Managing Parameters

The Gurobi optimizer provides a set of parameters to allow you to control many of the details of the optimization process. Factors like feasibility and optimality tolerances, choices of algorithms, strategies for exploring the MIP search tree, etc., can be controlled by modifying Gurobi parameters before beginning the optimization. Parameters are set using methods on a GRBEnv object (e.g., GRBEnv::set). Current values may also be retrieved with GRBEnv::get. Parameters can

be of type *int*, *double*, or *string*. You can also read a set of parameter settings from a file using GRBEnv::readParams, or write the set of changed parameters using GRBEnv::writeParams.

We also include an automated parameter tuning tool that explores many different sets of parameter changes in order to find a set that improves performance. You can call GRBModel::tune to invoke the tuning tool on a model. Refer to the parameter tuning tool section for more information.

One thing we should note is that each model gets its own copy of the environment when it is created. Parameter changes to the original environment therefore have no effect on existing models. Use GRBModel::getEnv to retrieve the environment associated with a particular model if you want to change a parameter for that model.

The full list of Gurobi parameters can be found in the Parameters section.

Memory Management

Memory management must always be considered in C++ programs. In particular, the Gurobi library and the user program share the same C++ heap, so the user must be aware of certain aspects of how the Gurobi library uses this heap. The basic rules for managing memory when using the Gurobi optimizer are as follows:

- As with other dynamically allocated C++ objects, GRBEnv or GRBModel objects should be freed using the associated destructors. In other words, given a `GRBModel` object `m`, you should call `delete m` when you are no longer using `m`.
- Objects that are associated with a model (e.g., GRBConstr, GRBSOS, and GRBVar objects) are managed by the model. In particular, deleting a model will delete all of the associated objects. Similarly, removing an object from a model (using GRBModel::remove) will also delete the object.
- Some Gurobi methods return an array of objects or values. For example, GRBModel::addVars returns an array of `GRBVar` objects. It is the user's responsibility to free the returned array (using `delete[]`). The reference manual indicates when a method returns a heap-allocated result.

One consequence of these rules is that you must be careful not to use an object once it has been freed. This is no doubt quite clear for environments and models, where you call the destructors explicitly, but may be less clear for constraints and variables, which are implicitly deleted when the associated model is deleted.

Monitoring Progress - Logging and Callbacks

Progress of the optimization can be monitored through Gurobi logging. By default, Gurobi will send output to the screen. A few simple controls are available for modifying the default logging behavior. If you would like to direct output to a file as well as to the screen, specify the log file name in the GRBEnv constructor. You can modify the LogFile parameter if you wish to redirect the log to a different file after creating the environment object. The frequency of logging output can be controlled with the DisplayInterval parameter, and logging can be turned off entirely with the OutputFlag parameter. A detailed description of the Gurobi log file can be found in the Logging section.

More detailed progress monitoring can be done through the GRBCallback class. The GRBModel::setCallback method allows you to receive a periodic callback from the Gurobi optimizer. You do this by sub-classing the GRBCallback abstract class, and writing your own `callback()`

method on this class. You can call GRBCallback::getDoubleInfo, GRBCallback::getIntInfo, GRBCallback::getStringInfo, or GRBCallback::getSolution from within the callback to obtain additional information about the state of the optimization.

Modifying Solver Behavior - Callbacks

Callbacks can also be used to modify the behavior of the Gurobi optimizer. The simplest control callback is GRBCallback::abort, which asks the optimizer to terminate at the earliest convenient point. Method GRBCallback::setSolution allows you to inject a feasible solution (or partial solution) during the solution of a MIP model. Methods GRBCallback::addCut and GRBCallback::addLazy allow you to add *cutting planes* and *lazy constraints* during a MIP optimization, respectively.

Error Handling

All of the methods in the Gurobi C++ library can throw an exception of type GRBException. When an exception occurs, additional information on the error can be obtained by retrieving the error code (using method `GRBException::getErrorCode`), or by retrieving the exception message (using method `GRBException::getMessage`). The list of possible error return codes can be found in the Error Codes section.

3.1 GRBEnv

Gurobi environment object. Gurobi models are always associated with an environment. You must create an environment before can you create and populate a model. You will generally only need a single environment object in your program.

The methods on environment objects are mainly used to manage Gurobi parameters (e.g., get, getParamInfo, set).

GRBEnv()

Constructor for `GRBEnv` object. If the constructor is called with no arguments, no log file will be written for the environment.

You have the option of constructing either a local environment, which solves Gurobi models on the local machine, or a client environment for a Gurobi compute server, which will solve Gurobi models on a server machine. For the latter, choose the signature that allows you to specify the names of the Gurobi compute servers and the priority of the associated job.

Note that the `GRBEnv` constructor will check the current working directory for a file named `gurobi.env`, and it will attempt to read parameter settings from this file if it exists. The file should be in PRM format (briefly, each line should contain a parameter name, followed by the desired value for that parameter).

In general, you should aim to create a single Gurobi environment object in your program, even if you plan to work with multiple models. Reusing one environment is much more efficient than creating and destroying multiple environments.

```
GRBEnv  GRBEnv ( )
```

Create a Gurobi environment (with logging disabled).

Return value:

An environment object (with no associated log file).

```
GRBEnv  GRBEnv (  const string&  logFileName )
```

Create a Gurobi environment (with logging enabled).

Arguments:

`logFileName`: The desired log file name.

Return value:

An environment object.

```
GRBEnv  GRBEnv (  const string&  logFileName,
                  const string&  computeserver,
                  int            port,
                  const string&  password,
                  int            priority,
                  double         timeout )
```

Create a client Gurobi environment on a compute server.

Arguments:

`logFileName`: The name of the log file for this environment. Pass an empty string for no log file.

`computeserver`: A comma-separated list of Gurobi compute servers. You can refer to compute server machines using their names or their IP addresses.

`port`: The port number used to connect to the compute server. You should pass a -1 value, which indicates that the default port should be used, unless your server administrator has changed our recommended port settings.

`password`: The password for gaining access to the specified compute servers. Pass an empty string if no password is required.

`priority`: The priority of the job. Priorities must be between -100 and 100, with a default value of 0 (by convention). Higher priority jobs are chosen from the server job queue before lower priority jobs. A job with priority 100 runs immediately, bypassing the job queue and ignoring the job limit on the server. You should exercise caution with priority 100 jobs, since they can severely overload a server, which can cause jobs to fail, and in extreme cases can cause the server to crash.

`timeout`: Job timeout (in seconds). If the job doesn't reach the front of the queue before the specified timeout, the constructor will throw a `JOB_REJECTED` exception. Use a negative value to indicate that the call should never timeout.

Return value:

An environment object.

GRBEnv::get()

Query the value of a parameter.

```
double  get ( GRB_DoubleParam  param )
```

Query the value of a double-valued parameter.

Arguments:

`param`: The parameter being queried. Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

Return value:

The current value of the requested parameter.

```
int  get ( GRB_IntParam  param )
```

Query the value of an int-valued parameter.

Arguments:

`param`: The parameter being queried. Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

Return value:

The current value of the requested parameter.

```
const string  get ( GRB_StringParam  param )
```

Query the value of a string-valued parameter.

Arguments:

param: The parameter being queried. Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

Return value:

The current value of the requested parameter.

GRBEnv::getErrorMsg()

Query the error message for the most recent exception associated with this environment.

```
const string  getErrorMsg ( )
```

Return value:

The error string.

GRBEnv::getParamInfo()

Obtain information about a parameter.

```
void  getParamInfo ( GRB_DoubleParam  param,
                     double*          valP,
                     double*          minP,
                     double*          maxP,
                     double*          *defP )
```

Obtain detailed information about a double parameter.

Arguments:

param: The parameter of interest. Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

valP: The current value of the parameter.

minP: The minimum allowed value of the parameter.

maxP: The maximum allowed value of the parameter.

defP: The default value of the parameter.

```
void  getParamInfo ( GRB_IntParam  param,
                     int*          valP,
                     int*          minP,
                     int*          maxP,
                     int*          defP )
```

Obtain detailed information about an integer parameter.

Arguments:

param: The parameter of interest. Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

valP: The current value of the parameter.

minP: The minimum allowed value of the parameter.

maxP: The maximum allowed value of the parameter.

defP: The default value of the parameter.

```
void  getParamInfo (  GRB_StringParam  param,
                      string*          valP,
                      string*          defP )
```

Obtain detailed information about a string parameter.

Arguments:

param: The parameter of interest. Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

valP: The current value of the parameter.

defP: The default value of the parameter.

GRBEnv::message()

Write a message to the console and the log file.

```
void  message (  const string&  message )
```

Arguments:

message: Print a message to the console and to the log file. Note that this call has no effect unless the `OutputFlag` parameter is set.

GRBEnv::readParams()

Read new parameter settings from a file.

Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

```
void  readParams (  const string&  paramfile )
```

Arguments:

paramfile: Name of the file containing parameter settings. Parameters should be listed one per line, with the parameter name first and the desired value second. For example:

```
# Gurobi parameter file
Threads 1
MIPGap 0
```

Blank lines and lines that begin with the hash symbol are ignored.

GRBEnv::resetParams()

Reset all parameters to their default values.

Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

```
void  resetParams ( )
```

GRBEnv::set()

Set the value of a parameter.

Important notes:

Note that a model gets its own copy of the environment when it is created. Changes to the original environment have no effect on the copy. Use GRBModel::getEnv to retrieve the environment associated with a model if you would like a parameter change to affect that model.

```
void  set ( GRB_DoubleParam  param,
            double           newvalue )
```

Set the value of a double-valued parameter.

Arguments:

param: The parameter being modified. Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

newvalue: The desired new value of the parameter.

```
void  set ( GRB_IntParam  param,
            int           newvalue )
```

Set the value of an int-valued parameter.

Arguments:

param: The parameter being modified. Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

newvalue: The desired new value of the parameter.

```
void  set ( GRB_StringParam  param,
            const string&    newvalue )
```

Set the value of a string-valued parameter.

Arguments:

param: The parameter being modified. Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

newvalue: The desired new value of the parameter.

```
void  set (  const string&  param,
             const string&  newvalue )
```

Set the value of any parameter using strings alone.

Arguments:

param: The name of the parameter being modified. Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

newvalue: The desired new value of the parameter.

GRBEnv::writeParams()

Write all non-default parameter settings to a file.

Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

```
void  writeParams (  const string&  paramfile )
```

Arguments:

paramfile: Name of the file to which non-default parameter settings should be written. The previous contents are overwritten.

3.2 GRBModel

Gurobi model object. Commonly used methods include addVar (adds a new decision variable to the model), addConstr (adds a new constraint to the model), optimize (optimizes the current model), and get (retrieves the value of an attribute).

GRBModel()

Constructor for `GRBModel`. The simplest version creates an empty model. You can then call addVar and addConstr to populate the model with variables and constraints. The more complex constructors can read a model from a file, or make a copy of an existing model.

```
GRBModel  GRBModel (  const GRBEnv&  env )
```

Model constructor.

Arguments:

`env`: Environment for new model.

Return value:

New model object. Model initially contains no variables or constraints.

```
GRBModel  GRBModel (  const GRBEnv&  env,
                      const string&  filename )
```

Read a model from a file. Note that the type of the file is encoded in the file name suffix. Valid suffixes are `.mps`, `.rew`, `.lp`, `.rlp`, `.ilp`, or `.opb`. The files can be compressed, so additional suffixes of `.zip`, `.gz`, `.bz2`, or `.7z` are accepted.

Arguments:

`env`: Environment for new model.

`modelname`: Name of the file containing the model.

Return value:

New model object.

```
GRBModel  GRBModel (  const GRBModel&  model )
```

Create a copy of an existing model.

Arguments:

`model`: Model to copy.

Return value:

New model object. Model is a clone of the input model.

GRBModel::addConstr()

Add a single linear constraint to a model. Multiple signatures are available.

```
GRBConstr  addConstr (  const GRBLinExpr&  lhsExpr,
                        char               sense,
                        const GRBLinExpr&  rhsExpr,
                        string             name="" )
```

Add a single linear constraint to a model.

Arguments:

lhsExpr: Left-hand side expression for new linear constraint.

sense: Sense for new linear constraint (GRB_LESS_EQUAL, GRB_EQUAL, or GRB_GREATER_EQUAL).

rhsExpr: Right-hand side expression for new linear constraint.

name (optional): Name for new constraint.

Return value:

New constraint object.

```
GRBConstr  addConstr (  const GRBLinExpr&  lhsExpr,
                        char               sense,
                        GRBVar             rhsVar,
                        string             name="" )
```

Add a single linear constraint to a model.

Arguments:

lhsExpr: Left-hand side expression for new linear constraint.

sense: Sense for new linear constraint (GRB_LESS_EQUAL, GRB_EQUAL, or GRB_GREATER_EQUAL).

rhsVar: Right-hand side variable for new linear constraint.

name (optional): Name for new constraint.

Return value:

New constraint object.

```
GRBConstr  addConstr (  const GRBLinExpr&  lhsExpr,
                        char               sense,
                        double             rhsVal,
                        string             name="" )
```

Add a single linear constraint to a model.

Arguments:

lhsExpr: Left-hand side expression for new linear constraint.

sense: Sense for new linear constraint (GRB_LESS_EQUAL, GRB_EQUAL, or GRB_GREATER_EQUAL).

rhsVal: Right-hand side value for new linear constraint.

name (optional): Name for new constraint.

Return value:

New constraint object.

```
GRBConstr  addConstr (  GRBVar  lhsVar,
                        char    sense,
                        GRBVar  rhsVar,
                        string  name="" )
```

Add a single linear constraint to a model.

Arguments:

lhsVar: Left-hand side variable for new linear constraint.

sense: Sense for new linear constraint (GRB_LESS_EQUAL, GRB_EQUAL, or GRB_GREATER_-EQUAL).

rhsVar: Right-hand side variable for new linear constraint.

name (optional): Name for new constraint.

Return value:

New constraint object.

```
GRBConstr  addConstr (  GRBVar  lhsVar,
                        char    sense,
                        double  rhsVal,
                        string  name="" )
```

Add a single linear constraint to a model.

Arguments:

lhsVar: Left-hand side variable for new linear constraint.

sense: Sense for new linear constraint (GRB_LESS_EQUAL, GRB_EQUAL, or GRB_GREATER_-EQUAL).

rhsVal: Right-hand side value for new linear constraint.

name (optional): Name for new constraint.

Return value:

New constraint object.

```
GRBConstr  addConstr (  GRBTempConstr&  tc,
                        string          name="" )
```

Add a single linear constraint to a model.

Arguments:

tc: Temporary constraint object, created using an overloaded comparison operator. See GRBTempConstr for more information.

name (optional): Name for new constraint.

Return value:

New constraint object.

GRBModel::addConstrs()

Add new linear constraints to a model.

We recommend that you build your model one constraint at a time (using addConstr), since it introduces no significant overhead and we find that it produces simpler code. Feel free to use these methods if you disagree, though.

```
GRBConstr*  addConstrs (  int  count )
```

Add `count` new linear constraints to a model.

Arguments:

`count`: Number of constraints to add to the model. The new constraints are all of the form `0 <= 0`.

Return value:

Array of new constraint objects. Note that the result is heap-allocated, and must be returned to the heap by the user.

```
GRBConstr*  addConstrs (  const GRBLinExpr*  lhsExprs,
                          const char*        senses,
                          const double*      rhsVals,
                          const string*      names,
                          int                count )
```

Add `count` new linear constraints to a model.

Arguments:

`lhsExprs`: Left-hand side expressions for the new linear constraints.

`senses`: Senses for new linear constraints (`GRB_LESS_EQUAL`, `GRB_EQUAL`, or `GRB_GREATER_EQUAL`).

`rhsVals`: Right-hand side values for the new linear constraints.

`names`: Names for new constraints.

`count`: Number of constraints to add.

Return value:

Array of new constraint objects. Note that the result is heap-allocated, and must be returned to the heap by the user.

GRBModel::addQConstr()

Add a quadratic constraint to a model. Multiple signatures are available.

Important note: the algorithms that Gurobi uses to solve quadratically constrained problems can only handle certain types of quadratic constraints. Constraints of the following forms are always accepted:

- $x^T Q x + q^T x \leq b$, where Q is Positive Semi-Definite (PSD)
- $x^T x \leq y^2$, where x is a vector of variables, and y is a non-negative variable (a Second-Order Cone)
- $x^T x \leq yz$, where x is a vector of variables, and y and z are non-negative variables (a rotated Second-Order Cone)

If you add a constraint that isn't in one of these forms (and Gurobi presolve is unable to transform the constraint into one of these forms), you'll get an error when you try to solve the model. Constraints where the quadratic terms only involve binary variables will always be transformed into one of these forms.

```
GRBQConstr  addQConstr (  const GRBQuadExpr&  lhsExpr,
                          char                sense,
                          const GRBQuadExpr&  rhsExpr,
                          string              name="" )
```

Add a quadratic constraint to a model.

Arguments:

lhsExpr: Left-hand side expression for new quadratic constraint.

sense: Sense for new quadratic constraint (GRB_LESS_EQUAL or GRB_GREATER_EQUAL).

rhsExpr: Right-hand side expression for new quadratic constraint.

name (optional): Name for new constraint.

Return value:

New quadratic constraint object.

```
GRBQConstr  addQConstr (  const GRBQuadExpr&  lhsExpr,
                          char                sense,
                          GRBVar              rhsVar,
                          string              name="" )
```

Add a quadratic constraint to a model.

Arguments:

lhsExpr: Left-hand side expression for new quadratic constraint.

sense: Sense for new quadratic constraint (GRB_LESS_EQUAL or GRB_GREATER_EQUAL).

rhsVar: Right-hand side variable for new quadratic constraint.

name (optional): Name for new constraint.

Return value:

New quadratic constraint object.

```
GRBQConstr  addQConstr (  GRBTempConstr&  tc,
                          string          name="" )
```

Add a quadratic constraint to a model.

Arguments:

tc: Temporary constraint object, created using an overloaded comparison operator. See GRBTempConstr for more information.

name (optional): Name for new constraint.

Return value:

New quadratic constraint object.

GRBModel::addRange()

Add a single range constraint to a model. A range constraint states that the value of the input expression must be between the specified **lower** and **upper** bounds in any solution.

Note that range constraints are stored internally as equality constraints. We add an extra variable to the model to capture the range information. Thus, the Sense attribute on a range constraint will always be GRB_EQUAL.

```
GRBConstr  addRange (  const GRBLinExpr&  expr,
                       double             lower,
                       double             upper,
                       string             name="" )
```

Arguments:

`expr`: Linear expression for new range constraint.

`lower`: Lower bound for linear expression.

`upper`: Upper bound for linear expression.

`name (optional)`: Name for new constraint.

Return value:

New constraint object.

GRBModel::addRanges()

Add new range constraints to a model. A range constraint states that the value of the input expression must be between the specified `lower` and `upper` bounds in any solution.

```
GRBConstr*  addRanges (  const GRBLinExpr*  exprs,
                         const double*      lower,
                         const double*      upper,
                         const string*      names,
                         int                count )
```

Arguments:

`exprs`: Linear expressions for the new range constraints.

`lower`: Lower bounds for linear expressions.

`upper`: Upper bounds for linear expressions.

`name`: Names for new range constraints.

`count`: Number of range constraints to add.

Return value:

Array of new constraint objects. Note that the result is heap-allocated, and must be returned to the heap by the user.

GRBModel::addSOS()

Add an SOS constraint to the model.

```
GRBSOS  addSOS (  const GRBVar*  vars,
                  const double*  weights,
                  int            len,
                  int            type )
```

Arguments:

`vars`: Array of variables that participate in the SOS constraint.

`weights`: Weights for the variables in the SOS constraint.

`len`: Number of members in the new SOS set (length of `vars` and `weights` arrays).

`type`: SOS type (can be `GRB_SOS_TYPE1` or `GRB_SOS_TYPE2`).

Return value:

New SOS constraint.

GRBModel::addVar()

Add a single decision variable to a model.

```
GRBVar  addVar (  double  lb,
                  double  ub,
                  double  obj,
                  char    type,
                  string  name="" )
```

Add a variable; non-zero entries will be added later.

Arguments:

lb: Lower bound for new variable.

ub: Upper bound for new variable.

obj: Objective coefficient for new variable.

type: Variable type for new variable (GRB_CONTINUOUS, GRB_BINARY, GRB_INTEGER, GRB_SEMICONT, or GRB_SEMIINT).

name (optional): Name for new variable.

Return value:

New variable object.

```
GRBVar  addVar (  double            lb,
                  double            ub,
                  double            obj,
                  char              type,
                  int               numnz,
                  const GRBConstr*  constrs,
                  const double*     coeffs,
                  string            name="" )
```

Add a variable, and the associated non-zero coefficients.

Arguments:

lb: Lower bound for new variable.

ub: Upper bound for new variable.

obj: Objective coefficient for new variable.

type: Variable type for new variable (GRB_CONTINUOUS, GRB_BINARY, GRB_INTEGER, GRB_SEMICONT, or GRB_SEMIINT).

numnz: Number of constraints in which this new variable participates.

constrs: Array of constraints in which the variable participates.

coeffs: Array of coefficients for each constraint in which the variable participates.

name (optional): Name for new variable.

Return value:

New variable object.

```
GRBVar  addVar (  double            lb,
                  double            ub,
                  double            obj,
                  char              type,
                  const GRBColumn&  col,
                  string            name="" )
```

Add a variable, and the associated non-zero coefficients.

Arguments:

`lb`: Lower bound for new variable.

`ub`: Upper bound for new variable.

`obj`: Objective coefficient for new variable.

`type`: Variable type for new variable (`GRB_CONTINUOUS`, `GRB_BINARY`, `GRB_INTEGER`, `GRB_SEMICONT`, or `GRB_SEMIINT`).

`col`: GRBColumn object for specifying a set of constraints to which new variable belongs.

`name (optional)`: Name for new variable.

Return value:

New variable object.

GRBModel::addVars()

Add new decision variables to a model.

```
GRBVar*  addVars (  int   count,
                    char  type=GRB_CONTINUOUS )
```

Add `count` new decision variables to a model. All associated attributes take their default values, except the variable `type`, which is specified as an argument.

Arguments:

`count`: Number of variables to add.

`type (optional)`: Variable type for new variables (`GRB_CONTINUOUS`, `GRB_BINARY`, `GRB_INTEGER`, `GRB_SEMICONT`, or `GRB_SEMIINT`).

Return value:

Array of new variable objects. Note that the result is heap-allocated, and must be returned to the heap by the user.

```
GRBVar*  addVars (  const double*  lb,
                    const double*  ub,
                    const double*  obj,
                    const char*    type,
                    const string*  names,
                    int            count )
```

Add `count` new decision variables to a model. This signature allows you to use arrays to hold the various variable attributes (lower bound, upper bound, etc.).

Arguments:

`lb`: Lower bounds for new variables. Can be `NULL`, in which case the variables get lower bounds of 0.0.

ub: Upper bounds for new variables. Can be NULL, in which case the variables get infinite upper bounds.

obj: Objective coefficients for new variables. Can be NULL, in which case the variables get objective coefficients of 0.0.

type: Variable types for new variables (GRB_CONTINUOUS, GRB_BINARY, GRB_INTEGER, GRB_SEMICONT, or GRB_SEMIINT). Can be NULL, in which case the variables are assumed to be continuous.

names: Names for new variables. Can be NULL, in which case all variables are given default names.

count: The number of variables to add.

Return value:

Array of new variable objects. Note that the result is heap-allocated, and must be returned to the heap by the user.

```
GRBVar*  addVars (  const double*     lb,
                    const double*     ub,
                    const double*     obj,
                    const char*       type,
                    const string*     names,
                    const GRBColumn*  cols,
                    int               count )
```

Add new decision variables to a model. This signature allows you to specify the set of constraints to which each new variable belongs using an array of GRBColumn objects.

Arguments:

lb: Lower bounds for new variables. Can be NULL, in which case the variables get lower bounds of 0.0.

ub: Upper bounds for new variables. Can be NULL, in which case the variables get infinite upper bounds.

obj: Objective coefficients for new variables. Can be NULL, in which case the variables get objective coefficients of 0.0.

type: Variable types for new variables (GRB_CONTINUOUS, GRB_BINARY, GRB_INTEGER, GRB_SEMICONT, or GRB_SEMIINT). Can be NULL, in which case the variables are assumed to be continuous.

names: Names for new variables. Can be NULL, in which case all variables are given default names.

cols: GRBColumn objects for specifying a set of constraints to which each new column belongs.

count: The number of variables to add.

Return value:

Array of new variable objects. Note that the result is heap-allocated, and must be returned to the heap by the user.

GRBModel::chgCoeff()

Change one coefficient in the model. The desired change is captured using a GRBVar object, a GRBConstr object, and a desired coefficient for the specified variable in the specified constraint. If

you make multiple changes to the same coefficient, the last one will be applied.

Note that the change won't take effect until the next call to GRBModel::optimize or GRBModel::update on that model.

```
void  chgCoeff (  GRBConstr  constr,
                  GRBVar     var,
                  double     newvalue )
```

Arguments:

constr: Constraint for coefficient to be changed.

var: Variable for coefficient to be changed.

newvalue: Desired new value for coefficient.

GRBModel::chgCoeffs()

Change a list of coefficients in the model. Each desired change is captured using a `GRBVar` object, a `GRBConstr` object, and a desired coefficient for the specified variable in the specified constraint. The entries in the input arrays each correspond to a single desired coefficient change. If you make multiple changes to the same coefficient, the last one will be applied.

Note that the changes won't take effect until the next call to GRBModel::optimize or GRBModel::update on that model.

```
void  chgCoeffs (  const GRBConstr*  constrs,
                   const GRBVar*     vars,
                   const double*     vals,
                   int               len )
```

Arguments:

constrs: Constraints for coefficients to be changed.

vars: Variables for coefficients to be changed.

vals: Desired new values for coefficients.

len: Number of coefficients to change (length of `vars`, `constrs`, and `vals` arrays).

GRBModel::computeIIS()

Compute an Irreducible Inconsistent Subsystem (IIS). An IIS is a subset of the constraints and variable bounds of the original model. If all constraints in the model except those in the IIS are removed, the model is still infeasible. However, further removing any one member of the IIS produces a feasible result.

This method populates the `IISCONSTR` and `IISQCONSTR` constraint attributes, the `IISSOS` SOS attribute, and the `IISLB`, and `IISUB` variable attributes. You can also obtain information about the results of the IIS computation by writing a `.ilp` format file (see GRBModel::write). This file contains only the IIS from the original model.

Note that this method can be used to compute IISs for both continuous and MIP models.

```
void  computeIIS ( )
```

GRBModel::discardConcurrentEnvs()

Discard concurrent environments for a model.

The concurrent environments created by getConcurrentEnv will be used by every subsequent call to the concurrent optimizer until the concurrent environments are discarded.

```
void  discardConcurrentEnvs ( )
```

GRBModel::feasRelax()

Modifies the `GRBModel` object to create a feasibility relaxation. Note that you need to call optimize on the result to compute the actual relaxed solution.

The feasibility relaxation is a model that, when solved, minimizes the amount by which the solution violates the bounds and linear constraints of the original model. This method provides a number of options for specifying the relaxation.

If you specify `relaxobjtype=0`, the objective of the feasibility relaxation is to minimize the sum of the weighted magnitudes of the bound and constraint violations. The `lbpen`, `ubpen`, and `rhspen` arguments specify the cost per unit violation in the lower bounds, upper bounds, and linear constraints, respectively.

If you specify `relaxobjtype=1`, the objective of the feasibility relaxation is to minimize the weighted sum of the squares of the bound and constraint violations. The `lbpen`, `ubpen`, and `rhspen` arguments specify the coefficients on the squares of the lower bound, upper bound, and linear constraint violations, respectively.

If you specify `relaxobjtype=2`, the objective of the feasibility relaxation is to minimize the weighted count of bound and constraint violations. The `lbpen`, `ubpen`, and `rhspen` arguments specify the cost of violating a lower bound, upper bound, and linear constraint, respectively.

To give an example, if a constraint with `rhspen` value `p` is violated by 2.0, it would contribute `2*p` to the feasibility relaxation objective for `relaxobjtype=0`, it would contribute `2*2*p` for `relaxobjtype=1`, and it would contribute `p` for `relaxobjtype=2`.

The `minrelax` argument is a boolean that controls the type of feasibility relaxation that is created. If `minrelax=false`, optimizing the returned model gives a solution that minimizes the cost of the violation. If `minrelax=true`, optimizing the returned model finds a solution that minimizes the original objective, but only from among those solutions that minimize the cost of the violation. Note that `feasRelax` must solve an optimization problem to find the minimum possible relaxation when `minrelax=true`, which can be quite expensive.

There are two signatures for this method. The more complex one takes a list of variables and constraints, as well as penalties associated with relaxing the corresponding lower bounds, upper bounds, and constraints. If a variable or constraint is not included in one of these lists, the associated bounds or constraints may not be violated. The simpler signature takes a pair of boolean arguments, `vrelax` and `crelax`, that indicate whether variable bounds and/or constraints can be violated. If `vrelax`/`crelax` is `true`, then every bound/constraint is allowed to be violated, respectively, and the associated cost is 1.0.

Note that this is a destructive method: it modifies the model on which it is invoked. If you don't want to modify your original model, use the GRBModel constructor to create a copy before invoking this method.

```
double  feasRelax (  int               relaxobjtype,
                     bool              minrelax,
                     int               vlen,
                     int               clen,
                     const GRBVar*     vars,
                     double*           lbpen,
                     double*           ubpen,
                     const GRBConstr*  constr,
                     double*           rhspen )
```

Create a feasibility relaxation model.

Arguments:

relaxobjtype: The cost function used when finding the minimum cost relaxation.

minrelax: The type of feasibility relaxation to perform.

vlen: The length of the list of variables whose bounds are allowed to be violated.

clen: The length of the list of linear contraints that are allowed to be violated.

vars: Variables whose bounds are allowed to be violated.

lbpen: Penalty for violating a variable lower bound. One entry for each variable in argument vars.

ubpen: Penalty for violating a variable upper bound. One entry for each variable in argument vars.

constr: Linear constraints that are allowed to be violated.

rhspen: Penalty for violating a linear constraint. One entry for each variable in argument constr.

Return value:

Zero if minrelax is false. If minrelax is true, the return value is the objective value for the relaxation performed. If the value is less than 0, it indicates that the method failed to create the feasibility relaxation.

```
double  feasRelax (  int   relaxobjtype,
                     bool  minrelax,
                     bool  vrelax,
                     bool  crelax )
```

Simplified method for creating a feasibility relaxation model.

Arguments:

relaxobjtype: The cost function used when finding the minimum cost relaxation.

minrelax: The type of feasibility relaxation to perform.

vrelax: Indicates whether variable bounds can be relaxed (with a cost of 1.0 for any violations.

crelax: Indicates whether linear constraints can be relaxed (with a cost of 1.0 for any violations.

Return value:

Zero if minrelax is false. If minrelax is true, the return value is the objective value for the relaxation performed. If the value is less than 0, it indicates that the method failed to create the feasibility relaxation.

GRBModel::fixedModel()

Create the fixed model associated with a MIP model. The MIP model must have a solution loaded (e.g., after a call to the optimize method). In the fixed model, each integer variable is fixed to the value that variable takes in the MIP solution.

```
GRBModel  fixedModel ( )
```

Return value:

Fixed model associated with calling object.

GRBModel::get()

Query the value(s) of an attribute. Use this method for scalar model attributes, or for arrays of constraint or variable attributes.

```
char*  get (  GRB_CharAttr   attr,
              const GRBVar*  vars,
              int            count )
```

Query a char-valued variable attribute for an array of variables.

Arguments:

attr: The attribute being queried.

vars: An array of variables whose attribute values are being queried.

count: The number of variable attributes to retrieve.

Return value:

The current values of the requested attribute for each input variable. Note that the result is heap-allocated, and must be returned to the heap by the user.

```
char*  get (  GRB_CharAttr      attr,
              const GRBConstr*  constrs,
              int               count )
```

Query a char-valued constraint attribute for an array of constraints.

Arguments:

attr: The attribute being queried.

constrs: An array of constraints whose attribute values are being queried.

count: The number of constraint attributes to retrieve.

Return value:

The current values of the requested attribute for each input constraint. Note that the result is heap-allocated, and must be returned to the heap by the user.

```
char*  get (  GRB_CharAttr       attr,
              const GRBQConstr*  qconstrs,
              int                count )
```

Query a char-valued quadratic constraint attribute for an array of quadratic constraints.

Arguments:

attr: The attribute being queried.
constrs: An array of quadratic constraints whose attribute values are being queried.
count: The number of quadratic constraint attributes to retrieve.

Return value:
The current values of the requested attribute for each input quadratic constraint. Note that the result is heap-allocated, and must be returned to the heap by the user.

```
double  get (  GRB_DoubleAttr  attr )
```

Query the value of a double-valued model attribute.

Arguments:
attr: The attribute being queried.

Return value:
The current value of the requested attribute.

```
double*  get (  GRB_DoubleAttr  attr,
                const GRBVar*   vars,
                int             count )
```

Query a double-valued variable attribute for an array of variables.

Arguments:
attr: The attribute being queried.
vars: An array of variables whose attribute values are being queried.
count: The number of variable attributes to retrieve.

Return value:
The current values of the requested attribute for each input variable. Note that the result is heap-allocated, and must be returned to the heap by the user.

```
double*  get (  GRB_DoubleAttr    attr,
                const GRBConstr*  constrs,
                int               count )
```

Query a double-valued constraint attribute for an array of constraints.

Arguments:
attr: The attribute being queried.
constrs: An array of constraints whose attribute values are being queried.
count: The number of constraint attributes to retrieve.

Return value:
The current values of the requested attribute for each input constraint. Note that the result is heap-allocated, and must be returned to the heap by the user.

```
double*  get (  GRB_DoubleAttr     attr,
                const GRBQConstr*  qconstrs,
                int                count )
```

Query a double-valued quadratic constraint attribute for an array of quadratic constraints.

Arguments:

`attr`: The attribute being queried.

`constrs`: An array of quadratic constraints whose attribute values are being queried.

`count`: The number of quadratic constraint attributes to retrieve.

Return value:

The current values of the requested attribute for each input quadratic constraint. Note that the result is heap-allocated, and must be returned to the heap by the user.

```
int  get ( GRB_IntAttr  attr )
```

Query the value of an int-valued model attribute.

Arguments:

`attr`: The attribute being queried.

Return value:

The current value of the requested attribute.

```
int*  get ( GRB_IntAttr    attr,
            const GRBVar*  vars,
            int            count )
```

Query an int-valued variable attribute for an array of variables.

Arguments:

`attr`: The attribute being queried.

`vars`: An array of variables whose attribute values are being queried.

`count`: The number of variable attributes to retrieve.

Return value:

The current values of the requested attribute for each input variable. Note that the result is heap-allocated, and must be returned to the heap by the user.

```
int*  get ( GRB_IntAttr       attr,
            const GRBConstr*  constrs,
            int               count )
```

Query an int-valued constraint attribute for an array of constraints.

Arguments:

`attr`: The attribute being queried.

`constrs`: An array of constraints whose attribute values are being queried.

`count`: The number of constraint attributes to retrieve.

Return value:

The current values of the requested attribute for each input constraint. Note that the result is heap-allocated, and must be returned to the heap by the user.

```
string  get ( GRB_StringAttr  attr )
```

Query the value of a string-valued model attribute.

Arguments:

attr: The attribute being queried.

Return value:

The current value of the requested attribute.

```
string*  get (  GRB_StringAttr  attr,
                const GRBVar*   vars,
                int             count )
```

Query a string-valued variable attribute for an array of variables.

Arguments:

attr: The attribute being queried.

vars: An array of variables whose attribute values are being queried.

count: The number of variable attributes to retrieve.

Return value:

The current values of the requested attribute for each input variable. Note that the result is heap-allocated, and must be returned to the heap by the user.

```
string*  get (  GRB_StringAttr    attr,
                const GRBConstr*  constrs,
                int               count )
```

Query a string-valued constraint attribute for an array of constraints.

Arguments:

attr: The attribute being queried.

constrs: An array of constraints whose attribute values are being queried.

count: The number of constraint attributes to retrieve.

Return value:

The current values of the requested attribute for each input constraint. Note that the result is heap-allocated, and must be returned to the heap by the user.

```
string*  get (  GRB_StringAttr     attr,
                const GRBQConstr*  qconstrs,
                int                count )
```

Query a string-valued quadratic constraint attribute for an array of quadratic constraints.

Arguments:

attr: The attribute being queried.

constrs: An array of quadratic constraints whose attribute values are being queried.

count: The number of quadratic constraint attributes to retrieve.

Return value:

The current values of the requested attribute for each input quadratic constraint. Note that the result is heap-allocated, and must be returned to the heap by the user.

GRBModel::getCoeff()

Query the coefficient of variable `var` in linear constraint `constr` (note that the result can be zero).

```
double  getCoeff (  GRBConstr  constr,
                    GRBVar     var )
```

Arguments:

`constr`: The requested constraint.

`var`: The requested variable.

Return value:

The current value of the requested coefficient.

GRBModel::getCol()

Retrieve the list of constraints in which a variable participates, and the associated coefficients. The result is returned as a GRBColumn object.

```
GRBColumn  getCol (  GRBVar  var )
```

Arguments:

`var`: The variable of interest.

Return value:

A GRBColumn object that captures the set of constraints in which the variable participates.

GRBModel::getConcurrentEnv()

Create/retrieve a concurrent environment for a model.

This method provides fine-grained control over the concurrent optimizer. By creating your own concurrent environments and setting appropriate parameters on these environments (e.g., the Method parameter), you can control exactly which strategies the concurrent optimizer employs. For example, if you create two concurrent environments, and set `Method` to primal simplex for one and dual simplex for the other, subsequent concurrent optimizer runs will use the two simplex algorithms rather than the default choices.

Note that you must create contiguously numbered concurrent environments, starting with `num=0`. For example, if you want three concurrent environments, they must be numbered 0, 1, and 2.

Once you create concurrent environments, they will be used for every subsequent concurrent optimization on that model. Use discardConcurrentEnvs to revert back to default concurrent optimizer behavior.

```
GRBEnv  getConcurrentEnv (  int  num )
```

Arguments:

`num`: The concurrent environment number.

Return value:

The concurrent environment for the model.

GRBModel::getConstrByName()

Retrieve a linear constraint from its name. If multiple linear constraints have the same name, this method chooses one arbitrarily.

```
GRBConstr  getConstrByName (  const string&  name )
```

Arguments:

name: The name of the desired linear constraint.

Return value:

The requested linear constraint.

GRBModel::getConstrs()

Retrieve an array of all linear constraints in the model.

```
GRBConstr*  getConstrs ( )
```

Return value:

An array of all linear constraints in the model. Note that this array is heap-allocated, and must be returned to the heap by the user.

GRBModel::getEnv()

Query the environment associated with the model. Note that each model makes its own copy of the environment when it is created. To change parameters for a model, for example, you should use this method to obtain the appropriate environment object.

```
GRBEnv  getEnv ( )
```

Return value:

The environment for the model.

GRBModel::getObjective()

Retrieve a quadratic model objective.

Note that the constant and linear portions of the objective can also be retrieved using the ObjCon and Obj attributes.

```
GRBQuadExpr  getObjective ( )
```

Return value:

The model objective.

GRBModel::getPWLObj()

Retrieve the piecewise-linear objective function for a variable. The return value gives the number of points that define the function, and the x and y arguments give the coordinates of the points, respectively. The x and y arguments must be large enough to hold the result. Call this method with `NULL` values for x and y if you just want the number of points.

Refer to the description of setPWLObj for additional information on what the values in x and y mean.

```
int  getPWLObj ( GRBVar    var,
                 double[]  x,
                 double[]  y )
```

Arguments:

var: The variable whose objective function is being retrieved.

x: The x values for the points that define the piecewise-linear function. These will always be in non-decreasing order.

y: The y values for the points that define the piecewise-linear function.

Return value:

The number of points that define the piecewise-linear objective function.

GRBModel::getQConstr()

Retrieve the left-hand side expression from a quadratic constraint. The result is returned as a GRBQuadExpr object.

```
GRBQuadExpr  getQConstr ( GRBQConstr  qconstr )
```

Arguments:

qconstr: The quadratic constraint of interest.

Return value:

A GRBQuadExpr object that captures the left-hand side of the quadratic constraint.

GRBModel::getQConstrs()

Retrieve an array of all quadratic constraints in the model.

```
GRBQConstr*  getQConstrs ( )
```

Return value:

An array of all quadratic constraints in the model. Note that this array is heap-allocated, and must be returned to the heap by the user.

GRBModel::getRow()

Retrieve a list of variables that participate in a constraint, and the associated coefficients. The result is returned as a GRBLinExpr object.

```
GRBLinExpr  getRow ( GRBConstr  constr )
```

Arguments:

`constr`: The constraint of interest.

Return value:

A GRBLinExpr object that captures the set of variables that participate in the constraint.

GRBModel::getSOS()

Retrieve the list of variables that participate in an SOS constraint, and the associated coefficients. The return value is the length of this list. If you would like to allocate space for the result before retrieving the result, call the method first with `NULL` array arguments to determine the appropriate array lengths.

```
int  getSOS ( GRBSOS   sos,
              GRBVar*  vars,
              double*  weights,
              int*     typeP )
```

Arguments:

`sos`: The SOS set of interest.

`vars`: A list of variables that participate in `sos`.

`weights`: The SOS weights for each participating variable.

`typeP`: The type of the SOS set (either `GRB_SOS_TYPE1` or `GRB_SOS_TYPE2`).

Return value:

The length of the result arrays.

GRBModel::getSOSs()

Retrieve an array of all SOS constraints in the model.

```
GRBSOS*  getSOSs ( )
```

Return value:

An array of all SOS constraints in the model. Note that this array is heap-allocated, and must be returned to the heap by the user.

GRBModel::getTuneResult()

Use this method to retrieve the results of a previous tune call. Calling this method with argument `n` causes tuned parameter set `n` to be copied into the model. Parameter sets are stored in order of decreasing quality, with parameter set 0 being the best. The number of available sets is stored in attribute TuneResultCount.

Once you have retrieved a tuning result, you can call optimize to use these parameter settings to optimize the model, or write to write the changed parameters to a `.prm` file.

Please refer to the parameter tuning section for details on the tuning tool.

```
void  getTuneResult (  int  n )
```

n: The index of the tuning result to retrieve. The best result is available as index 0. The number of stored results is available in attribute TuneResultCount.

GRBModel::getVarByName()

Retrieve a variable from its name. If multiple variables have the same name, this method chooses one arbitrarily.

```
GRBVar  getVarByName (  const string&  name )
```

Arguments:

name: The name of the desired variable.

Return value:

The requested variable.

GRBModel::getVars()

Retrieve an array of all variables in the model.

```
GRBVar*  getVars ( )
```

Return value:

An array of all variables in the model. Note that this array is heap-allocated, and must be returned to the heap by the user.

GRBModel::optimize()

Optimize the model. The algorithm used for the optimization depends on the model type (simplex or barrier for a continuous model; branch-and-cut for a MIP model). Upon successful completion, this method will populate the solution related attributes of the model. See the Attributes section for more information on attributes.

```
void  optimize ( )
```

GRBModel::optimizeasync()

Optimize a model asynchronously. This routine returns immediately. Your program can perform other computations while optimization proceeds in the background. To check the state of the asynchronous optimization, query the Status attribute for the model. A value of IN_PROGRESS indicates that the optimization has not yet completed. When you are done with your foreground tasks, you must call sync to sync your foreground program with the asynchronous optimization task.

Note that the set of Gurobi calls that you are allowed to make while optimization is running in the background is severely limited. Specifically, you can only perform attribute queries, and only for a few attributes (listed below). Any other calls on the running model, *or on any other models that were built within the same Gurobi environment*, will fail with error code OPTIMIZATION_IN_PROGRESS.

Note that there are no such restrictions on models built in other environments. Thus, for example, you could create multiple environments, and then have a single foreground program launch multiple simultaneous asynchronous optimizations, each in its own environment.

As already noted, you are allowed to query the value of the Status attribute while an asynchronous optimization is in progress. The other attributes that can be queried are: ObjVal, ObjBound, IterCount, NodeCount, and BarIterCount. In each case, the returned value reflects progress in the optimization to that point. Any attempt to query the value of an attribute not on this list will return an OPTIMIZATION_IN_PROGRESS error.

```
void  optimizeasync ( )
```

GRBModel::presolve()

Perform presolve on a model.

```
GRBModel  presolve ( )
```

Return value:

Presolved version of original model.

GRBModel::read()

This method is the general entry point for importing data from a file into a model. It can be used to read basis files for continuous models, start vectors for MIP models, or parameter settings. The type of data read is determined by the file suffix. File formats are described in the File Format section.

Note that this isn't the method to use if you want to read a new model from a file. For that, use the GRBModel constructor. One variant of the constructor takes the name of the file that contains the new model as its argument.

```
void  read (  const string&  filename )
```

Arguments:

filename: Name of the file to read. The suffix on the file must be either `.bas` (for an LP basis), `.mst` or `.sol` (for a MIP start), `.hnt` (for MIP hints), `.ord` (for a priority order), or `.prm` (for a parameter file). The suffix may optionally be followed by `.zip`, `.gz`, `.bz2`, or `.7z`.

GRBModel::remove()

Remove a variable, constraint, or SOS from a model.

```
void  remove ( GRBConstr  constr )
```

Remove a linear constraint from the model. Note that the constraint isn't actually removed from the model until the next call to GRBModel::optimize or GRBModel::update on that model.

Arguments:

constr: The linear constraint to remove.

```
void  remove ( GRBQConstr  qconstr )
```

Remove a quadratic constraint from the model. Note that the constraint isn't actually removed from the model until the next call to GRBModel::optimize or GRBModel::update on that model.

Arguments:

qconstr: The quadratic constraint to remove.

```
void  remove ( GRBSOS  sos )
```

Remove an SOS constraint from the model. Note that the SOS isn't actually removed from the model until the next call to GRBModel::optimize or GRBModel::update on that model.

Arguments:

sos: The SOS constraint to remove.

```
void  remove ( GRBVar  var )
```

Remove a variable from the model. Note that the variable isn't actually removed from the model until the next call to GRBModel::optimize or GRBModel::update on that model.

Arguments:

var: The variable to remove.

GRBModel::reset()

Reset the model to an unsolved state, discarding any previously computed solution information.

```
void  reset ( )
```

GRBModel::setCallback()

Set the callback object for a model. The `callback()` method on this object will be called periodically from the Gurobi solver. You will have the opportunity to obtain more detailed information about the state of the optimization from this callback. See the documentation for GRBCallback for additional information.

Note that a model can only have a single callback method, so this call will replace an existing callback. To disable a previously set callback, call this method with a `NULL` argument.

```
void  setCallback (  GRBCallback*  cb )
```

GRBModel::set()

Set the value(s) of an attribute. Use this method for scalar model attributes and for arrays of constraint or variable attributes.

```
void  set (  GRB_CharAttr   attr,
             const GRBVar*  vars,
             char*          newvalues,
             int            count )
```

Set a char-valued variable attribute for an array of variables.

Arguments:

attr: The attribute being modified.

vars: An array of variables whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input variable.

count: The number of variable attributes to set.

```
void  set (  GRB_CharAttr      attr,
             const GRBConstr*  constrs,
             char*             newvalues,
             int               count )
```

Set a char-valued constraint attribute for an array of constraints.

Arguments:

attr: The attribute being modified.

constrs: An array of constraints whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input constraint.

count: The number of constraint attributes to set.

```
void  set (  GRB_CharAttr       attr,
             const GRBQConstr*  qconstrs,
             char*              newvalues,
             int                count )
```

Set a char-valued quadratic constraint attribute for an array of quadratic constraints.

Arguments:

attr: The attribute being modified.

constrs: An array of quadratic constraints whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input quadratic constraint.

count: The number of quadratic constraint attributes to set.

```
void  set (  GRB_DoubleAttr  attr,
             double          newvalue )
```

Set the value of a double-valued model attribute.

Arguments:

attr: The attribute being modified.

newvalue: The desired new value for the attribute.

```
void  set (  GRB_DoubleAttr  attr,
             const GRBVar*   vars,
             double*         newvalues,
             int             count )
```

Set a double-valued variable attribute for an array of variables.

Arguments:

attr: The attribute being modified.

vars: An array of variables whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input variable.

count: The number of variable attributes to set.

```
void  set (  GRB_DoubleAttr    attr,
             const GRBConstr*  constrs,
             double*           newvalues,
             int               count )
```

Set a double-valued constraint attribute for an array of constraints.

Arguments:

attr: The attribute being modified.

constrs: An array of constraints whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input constraint.

count: The number of constraint attributes to set.

```
void  set (  GRB_DoubleAttr     attr,
             const GRBQConstr*  qconstrs,
             double*            newvalues,
             int                count )
```

Set a double-valued quadratic constraint attribute for an array of quadratic constraints.

Arguments:

attr: The attribute being modified.

constrs: An array of quadratic constraints whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input quadratic constraint.

count: The number of quadratic constraint attributes to set.

```
void  set (  GRB_IntAttr  attr,
             int          newvalue )
```

Set the value of an int-valued model attribute.

Arguments:

attr: The attribute being modified.

newvalue: The desired new value for the attribute.

```
void  set (  GRB_IntAttr    attr,
             const GRBVar*  vars,
             int*           newvalues,
             int            count )
```

Set an int-valued variable attribute for an array of variables.

Arguments:

attr: The attribute being modified.

vars: An array of variables whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input variable.

count: The number of variable attributes to set.

```
void  set (  GRB_IntAttr       attr,
             const GRBConstr*  constrs,
             int*              newvalues,
             int               count )
```

Set an int-valued constraint attribute for an array of constraints.

Arguments:

attr: The attribute being modified.

constrs: An array of constraints whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input constraint.

count: The number of constraint attributes to set.

```
void  set (  GRB_StringAttr  attr,
             string          newvalue )
```

Set the value of a string-valued model attribute.

Arguments:

attr: The attribute being modified.

newvalue: The desired new value for the attribute.

```
void  set (  GRB_StringAttr  attr,
             const GRBVar*   vars,
             string*         newvalues,
             int             count )
```

Set a string-valued variable attribute for an array of variables.

Arguments:

attr: The attribute being modified.

vars: An array of variables whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input variable.

count: The number of variable attributes to set.

```
void  set (  GRB_StringAttr    attr,
             const GRBConstr*  constrs,
             string*           newvalues,
             int               count )
```

Set a string-valued constraint attribute for an array of constraints.

Arguments:

attr: The attribute being modified.

constrs: An array of constraints whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input constraint.

count: The number of constraint attributes to set.

```
void  set (  GRB_StringAttr     attr,
             const GRBQConstr*  qconstrs,
             string*            newvalues,
             int                count )
```

Set a string-valued quadratic constraint attribute for an array of quadratic constraints.

Arguments:

attr: The attribute being modified.

constrs: An array of quadratic constraints whose attribute values are being modified.

newvalues: The desired new values for the attribute for each input quadratic constraint.

count: The number of quadratic constraint attributes to set.

GRBModel::setObjective()

Set the model objective equal to a linear or quadratic expression.

Note that you can also modify the linear portion of a model objective using the Obj variable attribute. If you wish to mix and match these two approaches, please note that this method replaces the entire existing objective, while the `Obj` attribute can be used to modify individual linear terms.

```
void  setObjective (  GRBLinExpr  linexpr,
                      int         sense=0 )
```

Arguments:

linexpr: New linear model objective.

sense (optional): Optimization sense (`GRB_MINIMIZE` for minimization, `GRB_MAXIMIZE` for maximization). Omit this argument to use the `ModelSense` attribute value to determine the sense.

```
void  setObjective (  GRBQuadExpr  quadexpr,
                      int          sense=0 )
```

Arguments:

quadexpr: New quadratic model objective.

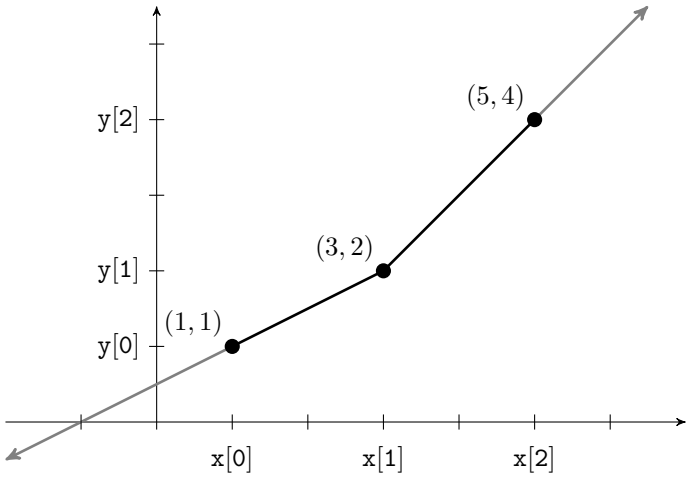
sense (optional): Optimization sense (GRB_MINIMIZE for minimization, GRB_MAXIMIZE for maximization). Omit this argument to use the ModelSense attribute value.

GRBModel::setPWLObj()

Set a piecewise-linear objective function for a variable.

The arguments to this method specify a list of points that define a piecewise-linear objective function for a single variable. Specifically, the x and y arguments give coordinates for the vertices of the function.

For example, suppose we want to define the function $f(x)$ shown below:



The vertices of the function occur at the points $(1, 1)$, $(3, 2)$ and $(5, 4)$, so **npoints** is 3, x is {1, 3, 5}, and y is {1, 2, 4}. With these arguments we define $f(1) = 1$, $f(3) = 2$ and $f(5) = 4$. Other objective values are linearly interpolated between neighboring points. The first pair and last pair of points each define a ray, so values outside the specified x values are extrapolated from these points. Thus, in our example, $f(-1) = 0$ and $f(6) = 5$.

More formally, a set of n points

$$\mathtt{x} = \{x_1, \ldots, x_n\}, \quad \mathtt{y} = \{y_1, \ldots, y_n\}$$

define the following piecewise-linear function:

$$f(v) = \begin{cases} y_1 + \frac{y_2 - y_1}{x_2 - x_1}(v - x_1), & \text{if } v \leq x_1, \\ y_i + \frac{y_{i+1} - y_i}{x_{i+1} - x_i}(v - x_i), & \text{if } v \geq x_i \text{ and } v \leq x_{i+1}, \\ y_n + \frac{y_n - y_{n-1}}{x_n - x_{n-1}}(v - x_n), & \text{if } v \geq x_n. \end{cases}$$

The x entries must appear in non-decreasing order. Two points can have the same x coordinate — this can be useful for specifying a discrete jump in the objective function.

Note that a piecewise-linear objective can change the type of a model. Specifically, including a non-convex piecewise linear objective function in a continuous model will transform that model into a MIP. This can significantly increase the cost of solving the model.

Setting a piecewise-linear objective for a variable will set the Obj attribute on that variable to 0. Similarly, setting the `Obj` attribute will delete the piecewise-linear objective on that variable.

Each variable can have its own piecewise-linear objective function. They must be specified individually, even if multiple variables share the same function.

```
void  setPWLObj ( GRBvar    var,
                  int       npoints,
                  double[]  x,
                  double[]  y )
```

Set the piecewise-linear objective function for a variable.

Arguments:

`var`: The variable whose objective function is being set.

`npoints`: Number of points that define the piecewise-linear function.

`x`: The x values for the points that define the piecewise-linear function. Must be in non-decreasing order.

`y`: The y values for the points that define the piecewise-linear function.

GRBModel::sync()

Wait for a previous asynchronous optimization call to complete.

Calling optimizeasync returns control to the calling routine immediately. The caller can perform other computations while optimization proceeds, and can check on the progress of the optimization by querying various model attributes. The `sync` call forces the calling program to wait until the asynchronous optimization call completes. You *must* call `sync` before the corresponding model object is deleted.

The `sync` call throws an exception if the optimization itself ran into any problems. In other words, exceptions thrown by this method are those that `optimize` itself would have thrown, had the original method not been asynchronous.

Note that you need to call `sync` even if you know that the asynchronous optimization has already completed.

```
void  sync ( )
```

GRBModel::terminate()

Generate a request to terminate the current optimization. This method can be called at any time during an optimization.

```
void  terminate ( )
```

GRBModel::tune()

Perform an automated search for parameter settings that improve performance. Upon completion, this method stores the best parameter sets it found. The number of stored parameter sets can be determined by querying the value of the TuneResultCount attribute. The actual settings can be retrieved using getTuneResult

Please refer to the parameter tuning section for details on the tuning tool.

```
void  tune ( )
```

GRBModel::update()

Process any pending model modifications.

```
void  update ( )
```

GRBModel::write()

This method is the general entry point for writing model data to a file. It can be used to write optimization models, IIS submodels, solutions, basis vectors, MIP start vectors, or parameter settings. The type of file written is determined by the file suffix. File formats are described in the File Format section.

```
void  write (  const string&  filename )
```

Arguments:

filename: Name of the file to write. The file type is encoded in the file name suffix. Valid suffixes for writing the model itself are `.mps`, `.rew`, `.lp`, or `.rlp`. An IIS can be written by using an `.ilp` suffix. Use `.sol` for a solution file, `.mst` for a MIP start, `.hnt` for MIP hints, `.bas` for a basis file, or `.prm` for a parameter file. The suffix may optionally be followed by `.gz`, `.bz2`, or `.7z`, which produces a compressed result.

3.3 GRBVar

Gurobi variable object. Variables are always associated with a particular model. You create a variable object by adding a variable to a model (using GRBModel::addVar), rather than by using a `GRBVar` constructor.

The methods on variable objects are used to get and set variable attributes. For example, solution information can be queried by calling get(`GRB_DoubleAttr_X`). Note that you can also query attributes for a set of variables at once. This is done using the attribute query method on the `GRBModel` object (GRBModel::get).

GRBVar::get()

Query the value of a variable attribute.

```
char  get (  GRB_CharAttr  attr )
```

Query the value of a char-valued attribute.
Arguments:
`attr`: The attribute being queried.
Return value:
The current value of the requested attribute.

```
double  get (  GRB_DoubleAttr  attr )
```

Query the value of a double-valued attribute.
Arguments:
`attr`: The attribute being queried.
Return value:
The current value of the requested attribute.

```
int  get (  GRB_IntAttr  attr )
```

Query the value of an int-valued attribute.
Arguments:
`attr`: The attribute being queried.
Return value:
The current value of the requested attribute.

```
string  get (  GRB_StringAttr  attr )
```

Query the value of a string-valued attribute.
Arguments:
`attr`: The attribute being queried.
Return value:
The current value of the requested attribute.

GRBVar::sameAs()

```
bool  sameAs ( GRBVar  var2 )
```

Check whether two variable objects refer to the same variable.

Arguments:

var2: The other variable.

Return value:

Boolean result indicates whether the two variable objects refer to the same model variable.

GRBVar::set()

Set the value of a variable attribute.

```
void  set ( GRB_CharAttr  attr,
            char          newvalue )
```

Set the value of a char-valued attribute.

Arguments:

attr: The attribute being modified.

newvalue: The desired new value of the attribute.

```
void  set ( GRB_DoubleAttr  attr,
            double          newvalue )
```

Set the value of a double-valued attribute.

Arguments:

attr: The attribute being modified.

newvalue: The desired new value of the attribute.

```
void  set ( GRB_IntAttr  attr,
            int          newvalue )
```

Set the value of an int-valued attribute.

Arguments:

attr: The attribute being modified.

newvalue: The desired new value of the attribute.

```
void  set ( GRB_StringAttr  attr,
            const string&   newvalue )
```

Set the value of a string-valued attribute.

Arguments:

attr: The attribute being modified.

newvalue: The desired new value of the attribute.

3.4 GRBConstr

Gurobi constraint object. Constraints are always associated with a particular model. You create a constraint object by adding a constraint to a model (using GRBModel::addConstr), rather than by using a `GRBConstr` constructor.

The methods on constraint objects are used to get and set constraint attributes. For example, constraint right-hand sides can be queried by calling get(`GRB_DoubleAttr_RHS`). Note that you can also query attributes for a set of constraints at once. This is done using the attribute query method on the `GRBModel` object (GRBModel::get).

GRBConstr::get()

Query the value of a constraint attribute.

```
char  get (  GRB_CharAttr  attr )
```

Query the value of a char-valued attribute.
Arguments:
`attr`: The attribute being queried.
Return value:
The current value of the requested attribute.

```
double  get (  GRB_DoubleAttr  attr )
```

Query the value of a double-valued attribute.
Arguments:
`attr`: The attribute being queried.
Return value:
The current value of the requested attribute.

```
int  get (  GRB_IntAttr  attr )
```

Query the value of an int-valued attribute.
Arguments:
`attr`: The attribute being queried.
Return value:
The current value of the requested attribute.

```
string  get (  GRB_StringAttr  attr )
```

Query the value of a string-valued attribute.
Arguments:
`attr`: The attribute being queried.
Return value:
The current value of the requested attribute.

GRBConstr::sameAs()

```
bool  sameAs ( GRBConstr  constr2 )
```

Check whether two constraint objects refer to the same constraint.

Arguments:

constr2: The other constraint.

Return value:

Boolean result indicates whether the two constraint objects refer to the same model constraint.

GRBConstr::set()

Set the value of a constraint attribute.

```
void  set ( GRB_CharAttr  attr,
            char          newvalue )
```

Set the value of a char-valued attribute.

Arguments:

attr: The attribute being modified.

newvalue: The desired new value of the attribute.

```
void  set ( GRB_DoubleAttr  attr,
            double          newvalue )
```

Set the value of a double-valued attribute.

Arguments:

attr: The attribute being modified.

newvalue: The desired new value of the attribute.

```
void  set ( GRB_IntAttr  attr,
            int          newvalue )
```

Set the value of an int-valued attribute.

Arguments:

attr: The attribute being modified.

newvalue: The desired new value of the attribute.

```
void  set ( GRB_StringAttr  attr,
            const string&   newvalue )
```

Set the value of a string-valued attribute.

Arguments:

attr: The attribute being modified.

newvalue: The desired new value of the attribute.

3.5 GRBQConstr

Gurobi quadratic constraint object. Quadratic constraints are always associated with a particular model. You create a quadratic constraint object by adding a constraint to a model (using GRBModel::addQConstr), rather than by using a `GRBQConstr` constructor.

The methods on quadratic constraint objects are used to get and set quadratic constraint attributes. For example, quadratic constraint right-hand sides can be queried by calling get(`GRB_DoubleAttr_QCRHS`). Note, however, that it is generally more efficient to query attributes for a set of constraints at once. This is done using the attribute query method on the `GRBModel` object (GRBModel::get).

GRBQConstr::get()

Query the value of a quadratic constraint attribute.

```
char  get ( GRB_CharAttr  attr )
```

Query the value of a char-valued attribute.
Arguments:
`attr`: The attribute being queried.
Return value:
The current value of the requested attribute.

```
double  get ( GRB_DoubleAttr  attr )
```

Query the value of a double-valued attribute.
Arguments:
`attr`: The attribute being queried.
Return value:
The current value of the requested attribute.

```
string  get ( GRB_StringAttr  attr )
```

Query the value of a string-valued attribute.
Arguments:
`attr`: The attribute being queried.
Return value:
The current value of the requested attribute.

GRBQConstr::set()

Set the value of a quadratic constraint attribute.

```
void  set ( GRB_CharAttr  attr,
            char          newvalue )
```

Set the value of a char-valued attribute.

Arguments:

attr: The attribute being modified.

newvalue: The desired new value of the attribute.

```
void  set (  GRB_DoubleAttr  attr,
             double          newvalue )
```

Set the value of a double-valued attribute.

Arguments:

attr: The attribute being modified.

newvalue: The desired new value of the attribute.

```
void  set (  GRB_StringAttr  attr,
             const string&   newvalue )
```

Set the value of a string-valued attribute.

Arguments:

attr: The attribute being modified.

newvalue: The desired new value of the attribute.

3.6 GRBSOS

Gurobi SOS constraint object. SOS constraints are always associated with a particular model. You create an SOS object by adding an SOS constraint to a model (using GRBModel::addSOS), rather than by using a `GRBSOS` constructor. Similarly, SOS constraints are removed using the GRBModel::remove method.

An SOS constraint can be of type 1 or 2 (`GRB_SOS_TYPE1` or `GRB_SOS_TYPE2`). A type 1 SOS constraint is a set of variables for which at most one variable in the set may take a value other than zero. A type 2 SOS constraint is an ordered set of variables where at most two variables in the set may take non-zero values. If two take non-zero values, they must be contiguous in the ordered set.

SOS constraint objects have one attribute, `IISSOS`, which can be queried with the GRBSOS::get method.

GRBSOS::get()

Query the value of an SOS attribute.

```
int  get ( GRB_IntAttr  attr )
```

Arguments:

`attr`: The attribute being queried.

Return value:

The current value of the requested attribute.

3.7 GRBExpr

Abstract base class for the GRBLinExpr and GRBQuadExpr classes. Expressions are used to build objectives and constraints. They are temporary objects that typically have short lifespans.

GRBExpr::getValue()

Compute the value of an expression for the current solution.

```
double  getValue ( )
```

Return value:

Value of the expression for the current solution.

3.8 GRBLinExpr

Gurobi linear expression object. A linear expression consists of a constant term, plus a list of coefficient-variable pairs that capture the linear terms. Linear expressions are used to build constraints. They are temporary objects that typically have short lifespans.

The `GRBLinExpr` class is a sub-class of the abstract base class GRBExpr.

You generally build linear expressions using overloaded operators. For example, if `x` is a GRBVar object, then `x + 1` is a GRBLinExpr object. Expressions can be built from constants (e.g., `expr = 0`), variables (e.g., `expr = 1 * x + 2 * y`), or from other expressions (e.g., `expr2 = 2 * expr1 + x`, or `expr3 = expr1 + 2 * expr2`). You can also modify existing expressions (e.g., `expr += x`, or `expr2 -= expr1`).

Another option for building expressions is to use the addTerms method, which adds an array of new terms at once. Terms can also be removed from an expression, using remove.

Note that the cost of building expressions depends heavily on the approach you use. While you can generally ignore this issue when building small expressions, you should be aware of a few efficiency issues when building large expressions:

- You should avoid using `expr = expr + x` in a loop. It will lead to runtimes that are quadratic in the number of terms in the expression.
- Using `expr += x` (or `expr -= x`) is much more efficient than `expr = expr + x`. Building a large expression by looping over `+=` statements is reasonably efficient, but it isn't the most efficient approach.
- The most efficient way to build a large expression is to make a single call to addTerms.

Individual terms in a linear expression can be queried using the getVar, getCoeff, and getConstant methods. You can query the number of terms in the expression using the size method.

Note that a linear expression may contain multiple terms that involve the same variable. These duplicate terms are merged when creating a constraint from an expression, but they may be visible when inspecting individual terms in the expression (e.g., when using getVar).

GRBLinExpr()

Linear expression constructor. Create a constant expression or an expression with one term.

```
GRBLinExpr  GRBLinExpr (  double  constant=0.0 )
```

Create a constant linear expression.

Arguments:

`constant (optional)`: Constant value for expression.

Return value:

A constant expression object.

```
GRBLinExpr  GRBLinExpr (  GRBVar  var,
                          double  coeff=1.0 )
```

Create an expression with one term.

Arguments:

`var`: Variable for expression term.

`coeff (optional)`: Coefficient for expression term.

Return value:

An expression object containing one linear term.

GRBLinExpr::addTerms()

Add new terms into a linear expression.

```
void  addTerms (  const double*  coeffs,
                  const GRBVar*  vars,
                  int            count )
```

Arguments:

`coeffs`: Coefficients for new terms.

`vars`: Variables for new terms.

`count`: Number of terms to add to the expression.

GRBLinExpr::clear()

Set a linear expression to 0.

You should use the overloaded `expr = 0` instead. The `clear` method is mainly included for consistency with our interfaces to non-overloaded languages.

```
void  clear ( )
```

GRBLinExpr::getConstant()

Retrieve the constant term from a linear expression.

```
double  getConstant ( )
```

Return value:

Constant from expression.

GRBLinExpr::getCoeff()

Retrieve the coefficient from a single term of the expression.

```
double  getCoeff (  int  i )
```

Arguments:

`i`: Index for coefficient of interest.

Return value:

Coefficient for the term at index `i` in the expression.

GRBLinExpr::getValue()

Compute the value of a linear expression for the current solution.

```
double  getValue ( )
```

Return value:

Value of the expression for the current solution.

GRBLinExpr::getVar()

Retrieve the variable object from a single term of the expression.

```
GRBVar  getVar (  int  i )
```

Arguments:

`i`: Index for term of interest.

Return value:

Variable for the term at index `i` in the expression.

GRBLinExpr::operator=

Set an expression equal to another expression.

```
GRBLinExpr  operator= (  const GRBLinExpr&  rhs )
```

Arguments:

`rhs`: Source expression.

Return value:

New expression object.

GRBLinExpr::operator+

Add one expression into another, producing a result expression.

```
GRBLinExpr  operator+ (  const GRBLinExpr&  rhs )
```

Arguments:

`rhs`: Expression to add.

Return value:

Expression object which is equal the sum of the invoking expression and the argument expression.

GRBLinExpr::operator-

Subtract one expression from another, producing a result expression.

```
GRBLinExpr  operator- (  const GRBLinExpr&  rhs )
```

Arguments:

rhs: Expression to subtract.

Return value:

Expression object which is equal the invoking expression minus the argument expression.

GRBLinExpr::operator+=

Add an expression into the invoking expression.

```
void  operator+= (  const GRBLinExpr&  expr )
```

Arguments:

expr: Expression to add.

GRBLinExpr::operator-=

Subtract an expression from the invoking expression.

```
void  operator-= (  const GRBLinExpr&  expr )
```

Arguments:

expr: Expression to subtract.

GRBLinExpr::operator*=

Multiply the invoking expression by a constant.

```
void  operator*= (  double  multiplier )
```

Arguments:

multiplier: Constant multiplier.

GRBLinExpr::remove()

Remove a term from a linear expression.

```
void  remove (  int  i )
```

Remove the term stored at index `i` of the expression.

Arguments:

`i`: The index of the term to be removed.

```
boolean  remove (  GRBVar  var )
```

Remove all terms associated with variable `var` from the expression.

Arguments:

`var`: The variable whose term should be removed.

Return value:

Returns `true` if the variable appeared in the linear expression (and was removed).

GRBLinExpr::size()

Retrieve the number of terms in the linear expression (not including the constant).

```
unsigned int  size ( )
```

Return value:

Number of terms in the expression.

3.9 GRBQuadExpr

Gurobi quadratic expression object. A quadratic expression consists of a linear expression, plus a list of coefficient-variable-variable triples that capture the quadratic terms. Quadratic expressions are used to build quadratic objective functions and quadratic constraints. They are temporary objects that typically have short lifespans.

The `GRBQuadExpr` class is a sub-class of the abstract base class GRBExpr.

You generally build quadratic expressions using overloaded operators. For example, if `x` is a GRBVar object, then `x * x` is a GRBQuadExpr object. Expressions can be built from constants (e.g., `expr = 0`), variables (e.g., `expr = 1 * x *x + 2 * x * y`), or from other expressions (e.g., `expr2 = 2 * expr1 + x * x`, or `expr3 = expr1 + 2 * expr2`). You can also modify existing expressions (e.g., `expr += x * x`, or `expr2 -= expr1`).

The other option for building expressions is to start with an empty expression (using the GRBQuadExpr constructor), and then add terms. Terms can be added individually (using addTerm) or in groups (using addTerms). Terms can also be removed from an expression (using remove).

Note that the cost of building expressions depends heavily on the approach you use. While you can generally ignore this issue when building small expressions, you should be aware of a few efficiency issues when building large expressions:

- You should avoid using `expr = expr + x*x` in a loop. It will lead to runtimes that are quadratic in the number of terms in the expression.
- Using `expr += x*x` (or `expr -= x*x`) is much more efficient than `expr = expr + x*x`. Building a large expression by looping over `+=` statements is reasonably efficient, but it isn't the most efficient approach.
- The most efficient way to build a large expression is to make a single call addTerms.

Individual terms in a quadratic expression can be queried using the getVar1, getVar2, and getCoeff methods. You can query the number of quadratic terms in the expression using the size method. To query the constant and linear terms associated with a quadratic expression, first obtain the linear portion of the quadratic expression using getLinExpr, and then use the getConstant, getCoeff, or getVar on the resulting GRBLinExpr object.

Note that a quadratic expression may contain multiple terms that involve the same variable pair. These duplicate terms are merged when creating the model objective from an expression, but they may be visible when inspecting individual terms in the expression (e.g., when using getVar1 and getVar2).

GRBQuadExpr()

Quadratic expression constructor. Create a constant expression or an expression with one term.

```
GRBQuadExpr  GRBQuadExpr (  double  constant=0.0 )
```

Create a constant quadratic expression.

Arguments:

`constant (optional)`: Constant value for expression.

Return value:

A constant expression object.

```
GRBQuadExpr  GRBQuadExpr (  GRBVar  var,
                            double  coeff=1.0 )
```

Create an expression with one term.

Arguments:

`var`: Variable for expression term.

`coeff (optional)`: Coefficient for expression term.

Return value:

An expression object containing one quadratic term.

```
GRBQuadExpr  GRBQuadExpr (  GRBLinExpr  linexpr )
```

Initialize a quadratic expression from an existing linear expression.

Arguments:

`orig`: Existing linear expression to copy.

Return value:

Quadratic expression object whose initial value is taken from the input linear expression.

GRBQuadExpr::addTerm()

Add a single new term into a quadratic expression.

```
void  addTerm (  double  coeff,
                 GRBVar  var )
```

Add a new linear term into a quadratic expression.

Arguments:

`coeff`: Coefficient for new linear term.

`var`: Variable for new linear term.

```
void  addTerm (  double  coeff,
                 GRBVar  var1,
                 GRBVar  var2 )
```

Add a new quadratic term into a quadratic expression.

Arguments:

`coeff`: Coefficient for new quadratic term.

`var1`: Variable for new quadratic term.

`var2`: Variable for new quadratic term.

GRBQuadExpr::addTerms()

Add new terms into a quadratic expression.

```
void  addTerms (  const double*  coeffs,
                  const GRBVar*  vars,
                  int            count )
```

Add new linear terms into a quadratic expression.

Arguments:

`coeffs`: Coefficients for new linear terms.

`vars`: Variables for new linear terms.

`count`: Number of linear terms to add to the quadratic expression.

```
void  addTerms (  const double*  coeffs,
                  const GRBVar*  vars1,
                  const GRBVar*  vars2,
                  int            count )
```

Add new quadratic terms into a quadratic expression.

Arguments:

`coeffs`: Coefficients for new quadratic terms.

`vars1`: First variables for new quadratic terms.

`vars2`: Second variables for new quadratic terms.

`count`: Number of quadratic terms to add to the quadratic expression.

GRBQuadExpr::clear()

Set a quadratic expression to 0.

You should use the overloaded `expr = 0` instead. The `clear` method is mainly included for consistency with our interfaces to non-overloaded languages.

```
void  clear ( )
```

GRBQuadExpr::getCoeff()

Retrieve the coefficient from a single quadratic term of the quadratic expression.

```
double  getCoeff (  int  i )
```

Arguments:

`i`: Index for coefficient of interest.

Return value:

Coefficient for the quadratic term at index `i` in the quadratic expression.

GRBQuadExpr::getLinExpr()

A quadratic expression is represented as a linear expression, plus a list of quadratic terms. This method retrieves the linear expression associated with the quadratic expression.

```
GRBLinExpr  getLinExpr ( )
```

Return value:

Linear expression associated with the quadratic expression.

GRBQuadExpr::getValue()

Compute the value of a quadratic expression for the current solution.

```
double  getValue ( )
```

Return value:

Value of the expression for the current solution.

GRBQuadExpr::getVar1()

Retrieve the first variable object associated with a single quadratic term from the expression.

```
GRBVar  getVar1 (  int  i )
```

Arguments:

`i`: Index for term of interest.

Return value:

First variable for the quadratic term at index `i` in the quadratic expression.

GRBQuadExpr::getVar2()

Retrieve the second variable object associated with a single quadratic term from the expression.

```
GRBVar  getVar2 (  int  i )
```

Arguments:

`i`: Index for term of interest.

Return value:

Second variable for the quadratic term at index `i` in the quadratic expression.

GRBQuadExpr::operator=

Set a quadratic expression equal to another quadratic expression.

```
GRBQuadExpr  operator= (  const GRBQuadExpr&  rhs )
```

Arguments:

rhs: Source quadratic expression.

Return value:

New quadratic expression object.

GRBQuadExpr::operator+

Add one expression into another, producing a result expression.

```
GRBQuadExpr  operator+ (  const GRBQuadExpr&  rhs )
```

Arguments:

rhs: Expression to add.

Return value:

Expression object which is equal the sum of the invoking expression and the argument expression.

GRBQuadExpr::operator-

Subtract one expression from another, producing a result expression.

```
GRBQuadExpr  operator- (  const GRBQuadExpr&  rhs )
```

Arguments:

rhs: Expression to subtract.

Return value:

Expression object which is equal the invoking expression minus the argument expression.

GRBQuadExpr::operator+=

Add an expression into the invoking expression.

```
void  operator+= (  const GRBQuadExpr&  expr )
```

Arguments:

expr: Expression to add.

GRBQuadExpr::operator-=

Subtract an expression from the invoking expression.

```
void  operator-= (  const GRBQuadExpr&  expr )
```

Arguments:

`expr`: Expression to subtract.

GRBQuadExpr::operator*=

Multiply the invoking expression by a constant.

```
void  operator*= (  double  multiplier )
```

Arguments:

`multiplier`: Constant multiplier.

GRBQuadExpr::remove()

Remove a quadratic term from a quadratic expression.

```
void  remove (  int  i )
```

Remove the quadratic term stored at index `i` of the expression.

Arguments:

`i`: The index of the term to be removed.

```
boolean  remove (  GRBVar  var )
```

Remove all quadratic terms associated with variable `var` from the quadratic expression.

Arguments:

`var`: The variable whose term should be removed.

Return value:

Returns `true` if the variable appeared in the quadratic expression (and was removed).

GRBQuadExpr::size()

Retrieve the number of quadratic terms in the quadratic expression.

```
unsigned int  size ( )
```

Return value:

Number of quadratic terms in the expression.

3.10 GRBTempConstr

Gurobi temporary constraint object. Objects of this class are created as intermediate results when building constraints using overloaded operators. There are no member functions on this class. Instead, `GRBTempConstr` objects are created by a set of non-member functions: ==, <=, and >=. You will generally never store objects of this class in your own variables.

Consider the following examples:

```
model.addConstr(x + y <= 1);
model.addQConstr(x*x + y*y <= 1);
```

The overloaded `<=` operator creates an object of type `GRBTempContr`, which is then immediately passed to method GRBModel::addConstr or GRBModel::addQConstr.

3.11 GRBColumn

Gurobi column object. A column consists of a list of coefficient, constraint pairs. Columns are used to represent the set of constraints in which a variable participates, and the associated coefficients. They are temporary objects that typically have short lifespans.

You generally build columns by starting with an empty column (using the GRBColumn constructor), and then adding terms. Terms can be added individually, using addTerm, or in groups, using addTerms. Terms can also be removed from a column, using remove.

Individual terms in a column can be queried using the getConstr, and getCoeff methods. You can query the number of terms in the column using the size method.

GRBColumn()

Column constructor. Create an empty column.

```
GRBColumn  GRBColumn ( )
```

Return value:

An empty column object.

GRBColumn::addTerm()

Add a single term into a column.

```
void  addTerm (  double     coeff,
                 GRBConstr  constr )
```

Arguments:

`coeff`: Coefficient for new term.

`constr`: Constraint for new term.

GRBColumn::addTerms()

Add new terms into a column.

```
void  addTerms (  const double*     coeffs,
                  const GRBConstr*  constrs,
                  int               count )
```

Add a list of terms into a column.

Arguments:

`coeffs`: Coefficients for new terms.

`constrs`: Constraints for new terms.

`count`: Number of terms to add to the column.

GRBColumn::clear()

Remove all terms from a column.

```
void  clear ( )
```

GRBColumn::getCoeff()

Retrieve the coefficient from a single term in the column.

```
double  getCoeff (  int  i )
```

Return value:
Coefficient for the term at index `i` in the column.

GRBColumn::getConstr()

Retrieve the constraint object from a single term in the column.

```
GRBConstr  getConstr (  int  i )
```

Return value:
Constraint for the term at index `i` in the column.

GRBColumn::remove()

Remove a single term from a column.

```
void  remove (  int  i )
```

Remove the term stored at index `i` of the column.
Arguments:
`i`: The index of the term to be removed.

```
boolean  remove (  GRBConstr  constr )
```

Remove the term associated with constraint `constr` from the column.
Arguments:
`constr`: The constraint whose term should be removed.
Return value:
Returns `true` if the constraint appeared in the column (and was removed).

GRBColumn::size()

Retrieve the number of terms in the column.

```
unsigned int  size ( )
```
Return value:
Number of terms in the column.

3.12 GRBCallback

Gurobi callback class. This is an abstract class. To implement a callback, you should create a subclass of this class and implement a `callback()` method. If you pass an object of this subclass to method GRBModel::setCallback before calling GRBModel::optimize, the `callback()` method of the class will be called periodically. Depending on where the callback is called from, you can obtain various information about the progress of the optimization.

Note that this class contains one protected *int* member variable: `where`. You can query this variable from your `callback()` method to determine where the callback was called from.

Gurobi callbacks can be used both to monitor the progress of the optimization and to modify the behavior of the Gurobi optimizer. A simple user callback function might call the GRBCallback::getIntInfo or GRBCallback::getDoubleInfo methods to produce a custom display, or perhaps to terminate optimization early (using GRBCallback::abort). More sophisticated MIP callbacks might use GRBCallback::getNodeRel or GRBCallback::getSolution to retrieve values from the solution to the current node, and then use GRBCallback::addCut or GRBCallback::addLazy to add a constraint to cut off that solution, or GRBCallback::setSolution to import a heuristic solution built from that solution.

When solving a model using multiple threads, note that the user callback is only ever called from a single thread, so you don't need to worry about the thread-safety of your callback.

You can look at the `callback_c++.cpp` example for details of how to use Gurobi callbacks.

GRBCallback()

Callback constructor.

```
GRBCallback  GRBCallback ( )
```

Return value:

A callback object.

GRBCallback::abort()

Abort optimization. When the optimization stops, the Status attribute will be equal to `GRB_INTERRUPTED`.

```
void  abort ( )
```

GRBCallback::addCut()

Add a cutting plane to the MIP model from within a callback function. Note that this method can only be invoked when the `where` member variable is equal to `GRB_CB_MIPNODE` (see the Callback Codes section for more information).

Cutting planes can be added at any node of the branch-and-cut tree. However, they should be added sparingly, since they increase the size of the relaxation model that is solved at each node and can significantly degrade node processing speed.

Cutting planes are typically used to cut off the current relaxation solution. To retrieve the relaxation solution at the current node, you should first call getNodeRel.

When adding your own cuts, you must set parameter PreCrush to value 1. This setting shuts off a few presolve reductions that sometimes prevent cuts on the original model from being applied to the presolved model.

Note that cutting planes added through this method must truly be cutting planes — they can cut off continuous solutions, but they may not cut off integer solutions that respect the original constraints of the model. Ignoring this restriction will lead to incorrect solutions.

```
void  addCut ( const GRBLinExpr&  lhsExpr,
               char               sense,
               double             rhsVal )
```

Arguments:

lhsExpr: Left-hand side expression for new cutting plane.

sense: Sense for new cutting plane (GRB_LESS_EQUAL, GRB_EQUAL, or GRB_GREATER_EQUAL).

rhsVal: Right-hand side value for new cutting plane.

```
void  addCut ( GRBTempConstr&  tc )
```

Arguments:

tc: Temporary constraint object, created using an overloaded comparison operator. See GRBTempConstr for more information.

GRBCallback::addLazy()

Add a lazy constraint to the MIP model from within a callback function. Note that this method can only be invoked when the **where** member variable is equal to GRB_CB_MIPNODE or GRB_CB_MIPSOL (see the Callback Codes section for more information).

Lazy constraints are typically used when the full set of constraints for a MIP model is too large to represent explicitly. By only including the constraints that are actually violated by solutions found during the branch-and-cut search, it is sometimes possible to find a proven optimal solution while only adding a fraction of the full set of constraints.

You would typically add a lazy constraint by first querying the current node solution (by calling getSolution from a GRB_CB_MIPSOL callback, or getNodeRel from a GRB_CB_MIPNODE callback), and then calling addLazy() to add a constraint that cuts off the solution. Gurobi guarantees that you will have the opportunity to cut off any solutions that would otherwise be considered feasible.

Your callback should be prepared to cut off solutions that violate any of your lazy constraints, including those that have already been added. Node solutions will usually respect previously added lazy constraints, but not always.

Note that you must set the LazyConstraints parameter if you want to use lazy constraints.

```
void  addLazy ( const GRBLinExpr&  lhsExpr,
                char               sense,
                double             rhsVal )
```

Arguments:

lhsExpr: Left-hand side expression for new lazy constraint.

sense: Sense for new lazy constraint (GRB_LESS_EQUAL, GRB_EQUAL, or GRB_GREATER_-EQUAL).
rhsVal: Right-hand side value for new lazy constraint.

```
void  addLazy ( GRBTempConstr&  tc )
```

Arguments:
tc: Temporary constraint object, created using an overloaded comparison operator. See GRBTempConstr for more information.

GRBCallback::getDoubleInfo()

Request double-valued callback information. The available information depends on the value of the **where** member. For information on possible values of **where**, and the double-valued information that can be queried for different values of **where**, please refer to the Callback section.

```
double  getDoubleInfo ( int  what )
```

Arguments:
what: Information requested (refer the list of Gurobi Callback Codes for possible values).
Return value:
Value of requested callback information.

GRBCallback::getIntInfo()

Request int-valued callback information. The available information depends on the value of the **where** member. For information on possible values of **where**, and the int-valued information that can be queried for different values of **where**, please refer to the Callback section.

```
int  getIntInfo ( int  what )
```

Arguments:
what: Information requested (refer to the list of Gurobi Callback Codes for possible values).
Return value:
Value of requested callback information.

GRBCallback::getNodeRel()

Retrieve values from the node relaxation solution at the current node. Only available when the **where** member variable is equal to GRB_CB_MIPNODE, and GRB_CB_MIPNODE_STATUS is equal to GRB_OPTIMAL.

```
double  getNodeRel ( GRBVar  v )
```

Arguments:

v: The variable whose value is desired.

Return value:

The value of the specified variable in the node relaxation for the current node.

```
double*  getNodeRel (  const GRBVar*  xvars,
                       int            len )
```

Arguments:

xvars: The list of variables whose values are desired.

len: The number of variables in the list.

Return value:

The values of the specified variables in the node relaxation for the current node. Note that the result is heap-allocated, and must be returned to the heap by the user.

GRBCallback::getSolution()

Retrieve values from the current solution vector. Only available when the **where** member variable is equal to GRB_CB_MIPSOL.

```
double  getSolution (  GRBVar  v )
```

Arguments:

v: The variable whose value is desired.

Return value:

The value of the specified variable in the current solution vector.

```
double*  getSolution (  const GRBVar*  xvars,
                        int            len )
```

Arguments:

xvars: The list of variables whose values are desired.

len: The number of variables in the list.

Return value:

The values of the specified variables in the current solution. Note that the result is heap-allocated, and must be returned to the heap by the user.

GRBCallback::getStringInfo()

Request string-valued callback information. The available information depends on the value of the **where** member. For information on possible values of **where**, and the string-valued information that can be queried for different values of **where**, please refer to the Callback section.

```
string  getStringInfo (  int  what )
```

Arguments:

what: Information requested (refer to the list of Gurobi Callback Codes for possible values).

Return value:

Value of requested callback information.

GRBCallback::setSolution()

Import solution values for a heuristic solution. Only available when the **where** member variable is equal to GRB_CB_MIPNODE.

When you specify a heuristic solution from a callback, variables initially take undefined values. You should use this method to specify variable values. You can make multiple calls to **setSolution** from one callback invocation to specify values for multiple sets of variables. At the end of the callback, if values have been specified for any variables, the Gurobi optimizer will try to compute a feasible solution from the specified values, possibly filling in values for variables whose values were left undefined.

```
void  setSolution (  GRBVar  v,
                     double  val )
```

Arguments:

v: The variable whose values is being set.

val: The value of the variable in the new solution.

```
void  setSolution (  const GRBVar*  xvars,
                     const double*  sol,
                     int            len )
```

Arguments:

xvars: The variables whose values are being set.

sol: The values of the variables in the new solution.

len: The number of variables.

3.13 GRBException

Gurobi exception object. Exceptions can be thrown by nearly every method in the Gurobi C++ API.

GRBException()

Exception constructor.

```
GRBException  GRBException (  int  errcode=0 )
```

Create a Gurobi exception.

Arguments:

errcode (optional): Error code for exception.

Return value:

An exception object.

```
GRBException  GRBException (  string  errmsg,
                              int     errcode=0 )
```

Create a Gurobi exception.

Arguments:

errmsg: Error message for exception.

errcode (optional): Error code for exception.

Return value:

An exception object.

GRBException::getErrorCode()

Retrieve the error code associated with a Gurobi exception.

```
int  getErrorCode ( )
```

Return value:

The error code associated with the exception.

GRBException::getMessage()

Retrieve the error message associated with a Gurobi exception.

```
const string  getMessage ( )
```

Return value:

The error message associated with the exception.

3.14 Non-Member Functions

Several Gurobi C++ interface functions aren't member functions on a particular object.

operator==

Create an equality constraint

```
GRBTempConstr  operator== (  GRBQuadExpr  lhsExpr,
                             GRBQuadExpr  rhsExpr )
```

Arguments:

lhsExpr: Left-hand side of equality constraint.

rhsExpr: Right-hand side of equality constraint.

Return value:

A constraint of type GRBTempConstr. The result is typically immediately passed to GRBModel::addConstr.

operator<=

Create an inequality constraint

```
GRBTempConstr  operator<= (  GRBQuadExpr  lhsExpr,
                             GRBQuadExpr  rhsExpr )
```

Arguments:

lhsExpr: Left-hand side of inequality constraint.

rhsExpr: Right-hand side of inequality constraint.

Return value:

A constraint of type GRBTempConstr. The result is typically immediately passed to GRBModel::addConstr or GRBModel::addQConstr.

operator>=

Create an inequality constraint

```
GRBTempConstr  operator>= (  GRBQuadExpr  lhsExpr,
                             GRBQuadExpr  rhsExpr )
```

Arguments:

lhsExpr: Left-hand side of inequality constraint.

rhsExpr: Right-hand side of inequality constraint.

Return value:

A constraint of type GRBTempConstr. The result is typically immediately passed to GRBModel::addConstr or GRBModel::addQConstr.

operator+

Overloaded operator on expression objects.

```
GRBLinExpr  operator+ (  const GRBLinExpr&  expr1,
                         const GRBLinExpr&  expr2 )
```

Add a pair of expressions.

Arguments:

expr1: First expression to be added.

expr2: Second expression to be added.

Return value:

Sum expression.

```
GRBLinExpr  operator+ (  const GRBLinExpr&  expr )
```

Allow plus sign to be used before an expression.

Arguments:

expr: Expression.

Return value:

Result expression.

```
GRBLinExpr  operator+ (  GRBVar  x,
                         GRBVar  y )
```

Add a pair of variables.

Arguments:

x: First variable to be added.

y: Second variable to be added.

Return value:

Sum expression.

```
GRBQuadExpr  operator+ (  const GRBQuadExpr&  expr1,
                          const GRBQuadExpr&  expr2 )
```

Add a pair of expressions.

Arguments:

expr1: First expression to be added.

expr2: Second expression to be added.

Return value:

Sum expression.

```
GRBQuadExpr  operator+ (  const GRBQuadExpr&  expr )
```

Allow plus sign to be used before an expression.

Arguments:

expr: Expression.

Return value:

Result expression.

operator-

Overloaded operator on expression objects.

```
GRBLinExpr  operator- (  const GRBLinExpr&  expr1,
                         const GRBLinExpr&  expr2 )
```

Subtract one expression from another.

Arguments:

expr1: Start expression.

expr2: Expression to be subtracted.

Return value:

Difference expression.

```
GRBLinExpr  operator- (  const GRBLinExpr&  expr )
```

Negate an expression.

Arguments:

expr: Expression.

Return value:

Negation of expression.

```
GRBQuadExpr  operator- (  const GRBQuadExpr&  expr1,
                          const GRBQuadExpr&  expr2 )
```

Subtract one expression from another.

Arguments:

expr1: Start expression.

expr2: Expression to be subtracted.

Return value:

Difference expression.

```
GRBQuadExpr  operator- (  const GRBQuadExpr&  expr )
```

Negate an expression.

Arguments:

expr: Expression.

Return value:

Negation of expression.

operator*

Overloaded operator on expression objects.

```
GRBLinExpr  operator* (  GRBVar  x,
                         double  a )
```

Multiply a variable and a constant.

Arguments:

`x`: Variable.

`a`: Constant multiplier.

Return value:

Expression that represents the result of multiplying the variable by a constant.

```
GRBLinExpr  operator* (  double  a,
                         GRBVar  x )
```

Multiply a variable and a constant.

Arguments:

`a`: Constant multiplier.

`x`: Variable.

Return value:

Expression that represents the result of multiplying the variable by a constant.

```
GRBLinExpr  operator* (  const GRBLinExpr&  expr,
                         double             a )
```

Multiply an expression and a constant.

Arguments:

`expr`: Expression.

`a`: Constant multiplier.

Return value:

Expression that represents the result of multiplying the expression by a constant.

```
GRBLinExpr  operator* (  double             a,
                         const GRBLinExpr&  expr )
```

Multiply an expression and a constant.

Arguments:

`a`: Constant multiplier.

`expr`: Expression.

Return value:

Expression that represents the result of multiplying the expression by a constant.

```
GRBQuadExpr  operator* (  const GRBQuadExpr&  expr,
                          double              a )
```

Multiply an expression and a constant.

Arguments:

`expr`: Expression.

`a`: Constant multiplier.

Return value:

Expression that represents the result of multiplying the expression by a constant.

```
GRBQuadExpr  operator* (  double              a,
                          const GRBQuadExpr&  expr )
```

Multiply an expression and a constant.

Arguments:

`a`: Constant multiplier.

`expr`: Expression.

Return value:

Expression that represents the result of multiplying the expression by a constant.

```
GRBQuadExpr  operator* (  GRBVar  x,
                          GRBVar  y )
```

Multiply a pair of variables.

Arguments:

`x`: First variable.

`y`: Second variable.

Return value:

Expression that represents the result of multiplying the argument variables.

```
GRBQuadExpr  operator* (  GRBVar              var,
                          const GRBLinExpr&   expr )
```

Multiply an expression and a variable.

Arguments:

`var`: Variable.

`expr`: Expression.

Return value:

Expression that represents the result of multiplying the expression by a variable.

```
GRBQuadExpr  operator* (  const GRBLinExpr&  expr,
                          GRBVar             var )
```

Multiply an expression and a variable.

Arguments:

`var`: Variable.

`expr`: Expression.

Return value:

Expression that represents the result of multiplying the expression by a variable.

```
GRBQuadExpr  operator* (  const GRBLinExpr&  expr1,
                          const GRBLinExpr&  expr2 )
```

Multiply a pair of expressions.

Arguments:

`expr1`: First expression.

`expr2`: Second expression.

Return value:

Expression that represents the result of multiplying the argument expressions.

operator/

Overloaded operator to divide a variable or expression by a constant.

```
GRBLinExpr  operator/ (  GRBVar  x,
                         double  a )
```

Arguments:

`x`: Variable.

`a`: Constant divisor.

Return value:

Expression that represents the result of dividing the variable by a constant.

```
GRBLinExpr  operator/ (  const GRBLinExpr&  expr,
                         double             a )
```

Arguments:

`expr`: Expression.

`a`: Constant divisor.

Return value:

Expression that represents the result of dividing the expression by a constant.

```
GRBLinExpr  operator/ (  const GRBQuadExpr&  expr,
                         double              a )
```

Arguments:

`expr`: Expression.

`a`: Constant divisor.

Return value:

Expression that represents the result of dividing the expression by a constant.

3.15 Attribute Enums

These `enums` are used to get or set Gurobi attributes. The complete list of attributes can be found in the Attributes section.

GRB_CharAttr

This enum is used to get or set char-valued attributes (through GRBModel::get or GRBModel::set). Please refer to the Attributes section to see a list of all char attributes and their functions.

GRB_DoubleAttr

This enum is used to get or set double-valud attributes (through GRBModel::get or GRBModel::set). Please refer to the Attributes section to see a list of all double attributes and their functions.

GRB_IntAttr

This enum is used to get or set int-valued attributes (through GRBModel::get or GRBModel::set). Please refer to the Attributes section to see a list of all int attributes and their functions.

GRB_StringAttr

This enum is used to get or set string-valued attributes (through GRBModel::get or GRBModel::set). Please refer to the Attributes section to see a list of all string attributes and their functions.

3.16 Parameter Enums

These `enums` are used to get or set Gurobi parameters. The complete of parameters can be found in the Parameters section.

GRB_DoubleParam

This enum is used to get or set double-valued parameters (through GRBEnv::get or GRBEnv::set). Please refer to the Parameters section to see a list of all double parameters and their functions.

GRB_IntParam

This enum is used to get or set int-valued parameters (through GRBEnv::get or GRBEnv::set). Please refer to the Parameters section to see a list of all int parameters and their functions.

GRB_StringParam

This enum is used to get or set string-valued parameters (through GRBEnv::get or GRBEnv::set). Please refer to the Parameters section to see a list of all int parameters and their functions.