# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>About This Manual</td>
<td>6</td>
</tr>
<tr>
<td>Concepts</td>
<td>8</td>
</tr>
<tr>
<td>Group optim.cplex.cpp</td>
<td>12</td>
</tr>
<tr>
<td>ILOBARRIERCALLBACK0</td>
<td>18</td>
</tr>
<tr>
<td>ILOBRANCHCALLBACK0</td>
<td>19</td>
</tr>
<tr>
<td>ILOCONTINUOUCALLBACK0</td>
<td>20</td>
</tr>
<tr>
<td>ILOCPLEXGOAL0</td>
<td>21</td>
</tr>
<tr>
<td>ILOCROSSOVERCALLBACK0</td>
<td>23</td>
</tr>
<tr>
<td>ILOCUTCALLBACK0</td>
<td>24</td>
</tr>
<tr>
<td>ILODISJUNCTIVECUTCALLBACK0</td>
<td>25</td>
</tr>
<tr>
<td>ILOFRACTIONALCUTCALLBACK0</td>
<td>26</td>
</tr>
<tr>
<td>ILOHEURISTICCALLBACK0</td>
<td>27</td>
</tr>
<tr>
<td>ILOINCUMBENTCALLBACK0</td>
<td>28</td>
</tr>
<tr>
<td>ILOLAZYCONSTRAINTCALLBACK0</td>
<td>29</td>
</tr>
<tr>
<td>ILOMIPCALLBACK0</td>
<td>30</td>
</tr>
<tr>
<td>ILONETWORKCALLBACK0</td>
<td>31</td>
</tr>
<tr>
<td>ILONODECALLBACK0</td>
<td>32</td>
</tr>
<tr>
<td>ILOPRESOLVECALLBACK0</td>
<td>33</td>
</tr>
<tr>
<td>ILOPROBINGCALLBACK0</td>
<td>34</td>
</tr>
<tr>
<td>ILOSIMPLEXCALLBACK0</td>
<td>35</td>
</tr>
<tr>
<td>ILOSOLVECALLBACK0</td>
<td>36</td>
</tr>
<tr>
<td>ILOUSERCUTCALLBACK0</td>
<td>37</td>
</tr>
<tr>
<td>IloCplex</td>
<td>38</td>
</tr>
<tr>
<td>IloCplex::Algorithm</td>
<td>76</td>
</tr>
<tr>
<td>IloCplex::BarrierCallbackI</td>
<td>78</td>
</tr>
<tr>
<td>IloCplex::BasisStatus</td>
<td>80</td>
</tr>
<tr>
<td>IloCplex::BasisStatusArray</td>
<td>81</td>
</tr>
<tr>
<td>IloCplex::BoolParam</td>
<td>82</td>
</tr>
<tr>
<td>IloCplex::BranchCallbackI</td>
<td>84</td>
</tr>
<tr>
<td>BranchCallbackI::BranchType</td>
<td>93</td>
</tr>
<tr>
<td>IloCplex::BranchDirection</td>
<td>94</td>
</tr>
</tbody>
</table>
TABLE OF CONTENTS

IloCplex::BranchDirectionArray ........................................... 95
IloCplex::Callback ............................................................. 96
IloCplex::CallbackI ............................................................ 98
Callback::Type ............................................................... 102
IloCplex::ContinuousCallbackI ................................................. 103
IloCplex::ControlCallbackI .................................................... 106
ControlCallbackI::IntegerFeasibility ........................................ 114
IloCplex::CplexStatus .......................................................... 115
IloCplex::CrossoverCallbackI .................................................. 118
IloCplex::CutCallbackI ........................................................ 120
IloCplex::DeleteMode .......................................................... 123
IloCplex::DisjunctiveCutCallbackI ............................................. 124
IloCplex::DualPricing .......................................................... 126
IloCplex::Exception ............................................................ 127
IloCplex::FlowMIRCutCallbackI ................................................. 128
IloCplex::FractionalCutCallbackI .............................................. 130
IloCplex::Goal ................................................................. 132
IloCplex::GoalI ................................................................. 134
GoalI::BranchType ............................................................. 148
GoalI::IntegerFeasibility ...................................................... 149
GoalI::IntegerFeasibilityArray ................................................. 150
IloCplex::HeuristicCallbackI ................................................... 151
IloCplex::IISStatus ............................................................ 157
IloCplex::IISStatusArray ...................................................... 158
IloCplex::IncumbentCallbackI ................................................... 159
IloCplex::IntParam ............................................................. 162
IloCplex::InvalidCutException ............................................... 170
IloCplex::LazyConstraintCallbackI .......................................... 171
IloCplex::MIPCallbackI ....................................................... 173
MIPCallbackI::NodeData ........................................................ 180
IloCplex::MIPEmphasisType .................................................... 181

ILOG CPLEX C++ API 9.0 REFERENCE MANUAL
# Table of Contents

- `IloCplex::MultipleConversionException` ........................................... 183
- `IloCplex::MultipleObjException` .................................................. 184
- `IloCplex::NetworkCallbackI` ......................................................... 185
- `IloCplex::NodeCallbackI` .............................................................. 187
- `IloCplex::NodeEvaluator` ............................................................... 192
- `IloCplex::NodeEvaluatorI` ............................................................. 194
- `IloCplex::NodeSelect` ................................................................. 198
- `IloCplex::NumParam` .................................................................... 199
- `IloCplex::PWLFormulation` ............................................................ 202
- `IloCplex::PresolveCallbackI` ......................................................... 203
- `IloCplex::PrimalPricing` ............................................................... 205
- `IloCplex::ProbingCallbackI` ......................................................... 206
- `IloCplex::Quality` ...................................................................... 209
- `IloCplex::SearchLimit` ............................................................... 213
- `IloCplex::SearchLimitI` ............................................................... 215
- `IloCplex::SimplexCallbackI` ......................................................... 217
- `IloCplex::SolveCallbackI` ............................................................. 219
- `IloCplex::StringParam` ............................................................... 223
- `IloCplex::UnknownExtractableException` .................................... 224
- `IloCplex::UserCutCallbackI` ......................................................... 225
- `IloCplex::VariableSelect` ............................................................. 227

**Group optim.cplex.cpp.piecewiselinear** ........................................... 228

- `IloCplex::PWLFormulation` ............................................................ 230

**Group optim.cplex.cpp.logical** .................................................... 230

**Group optim.cplex.advanced.cpp** .................................................. 231

- `IloCplex::LazyConstraintCallbackI` ............................................. 233
- `IloCplex::UserCutCallbackI` ......................................................... 235
About This Manual

This reference manual documents the application programming interface (API) of ILOG CPLEX. There are separate reference manuals for the Callable Library, C++, Java, and C#.NET APIs.

What Are the ILOG CPLEX Component Libraries?

The ILOG CPLEX Component Libraries are designed to facilitate the development of applications to solve, modify, and interpret the results of linear, mixed integer, continuous convex quadratic, quadratically constrained, and mixed integer quadratic or quadratically constrained programming.

The ILOG CPLEX Component Libraries consist of:

- the CPLEX Callable Library, a C application programming interface (API), and
- ILOG Concert Technology, an object-oriented API for C++, Java, and C#.NET users.

ILOG Concert Technology is also part of ILOG Solver, enabling cooperative strategies using CPLEX and Solver together for solving difficult optimization problems.

What You Need to Know

This manual assumes that you are familiar with the operating system on which you are using ILOG CPLEX.

The CPLEX Callable Library is written in the C programming language. The Concert Technology libraries are written in the C++, Java, and C#.NET programming languages. If you use these products, this manual assumes you can write code in the appropriate language, and that you have a working knowledge of a supported integrated development environment (IDE) for that language.

Notation and Naming Conventions

Throughout this manual:
◆ The names of routines and parameters defined in the CPLEX Callable Library begin with CPX. The names of classes and functions defined in Concert Technology C++ API begin with Ilo. This convention helps prevent name space conflicts with user-written routines and other code libraries.

◆ The names of Component Library classes, functions, routines, and routine arguments appear in this typeface (examples: CPXprimopt, numcols, IloCplex::solveFixed).

Related Documentation

In addition to this Reference Manual documenting C and C++, ILOG CPLEX also comes with these resources:

◆ Getting Started with ILOG CPLEX introduces you to ways of specifying models and solving problems with ILOG CPLEX.

◆ The ILOG CPLEX User's Manual explores programming with ILOG CPLEX in greater depth. It provides practical ideas about how to use CPLEX in your own applications and shows how and why design and implementation decisions in the examples were made.

◆ The ILOG CPLEX Release Notes highlight the new features and important changes in this version.

◆ The ILOG CPLEX Java Reference Manual supplies detailed definitions of the Concert Technology Java interfaces and ILOG CPLEX Java classes.

◆ The ILOG CPLEX C#.NET Reference Manual supplies detailed definitions of the Concert Technology C#.NET interfaces and ILOG CPLEX C#.NET classes.

◆ Source code for examples is delivered in the standard distribution.

◆ A readme.html file is delivered in the standard distribution. This file contains the most current information about platform prerequisites for ILOG CPLEX.

All of the manuals and Release Notes are available in online versions. The online documentation, in HTML format, can be accessed through standard HTML browsers.
IloCplex, like all CPLEX technologies, uses branch & cut search when solving mixed integer programming (MIP) models. The branch & cut search procedure manages a search tree consisting of nodes. Every node represents an LP or QP subproblem to be processed; that is, to be solved, to be checked for integrality, and perhaps to be analyzed further. Nodes are called active if they have not yet been processed. After a node has been processed, it is no longer active. IloCplex processes active nodes in the tree until either no more active nodes are available or some limit has been reached.

A branch is the creation of two new nodes from a parent node. Typically, a branch occurs when the bounds on a single variable are modified, with the new bounds remaining in effect for that new node and for any of its descendants. For example, if a branch occurs on a binary variable, that is, one with a lower bound of 0 (zero) and an upper bound of 1 (one), then the result will be two new nodes, one node with a modified upper bound of 0 (the downward branch, in effect requiring this variable to take only the value 0), and the other node with a modified lower bound of 1 (the upward branch, placing the variable at 1). The two new nodes will thus have completely distinct solution domains.

A cut is a constraint added to the model. The purpose of adding any cut is to limit the size of the solution domain for the continuous LP or QP problems represented at the nodes, while not eliminating legal integer solutions. The outcome is thus to reduce the number of branches required to solve the MIP.

As an example of a cut, first consider the following constraint involving three binary (0-1) variables:

$$20x + 25y + 30z \leq 40$$

That sample constraint can be strengthened by adding the following cut to the model:

$$1x + 1y + 1z \leq 1$$
No feasible integer solutions are ruled out by the cut, but some fractional solutions, for example \((0.0, 0.4, 1.0)\), can no longer be obtained in any LP or QP subproblems at the nodes, possibly reducing the amount of searching needed.

The branch & cut method, then, consists of performing branches and applying cuts at the nodes of the tree. Here is a more detailed outline of the steps involved.

First, the branch & cut tree is initialized to contain the root node as the only active node. The root node of the tree represents the entire problem, ignoring all of the explicit integrality requirements. Potential cuts are generated for the root node but, in the interest of keeping the problem size reasonable, not all such cuts are applied to the model immediately. An incumbent solution, that is, the best known solution that satisfies all the integrality requirements, is established at this point for later use in the algorithm. Ordinarily, no such solution is available at this stage, but the user can specify a starting solution using the method `setVectors`. Alternatively, when solving a sequence of modified problems, the user can set the parameter `MIPStart` to the value 1 (one) to indicate that the solution of the previous problem should be used as a starting incumbent for the present problem.

When processing a node, `IloCplex` starts by solving the continuous relaxation of its subproblem. that is, the subproblem without integrality constraints. If the solution violates any cuts, `IloCplex` may add some or all of them to the node problem and resolves it. This procedure is iterated until no more violated cuts are detected (or deemed worth adding at this time) by the algorithm. If at any point in the addition of cuts the node becomes infeasible, the node is pruned (that is, it is removed from the tree).

Otherwise, `IloCplex` checks whether the solution of the node-problem satisfies the integrality constraints. If so, and if its objective value is better than that of the current incumbent, the solution of the node-problem is used as the new incumbent. If not, branching will occur, but first a heuristic method may be tried at this point to see if a new incumbent can be inferred from the LP/QP solution at this node, and other methods of analysis may be performed on this node. The branch, when it occurs, is performed on a variable where the value of the present solution violates its integrality requirement. This practice results in two new nodes being added to the tree for later processing.

Each node, after its relaxation is solved, possesses an optimal objective function value \(Z\). At any given point in the algorithm, there is a node whose \(Z\) value is better (less, in the case of a minimization problem, or greater for a maximization problem) than all the others. This Best Node value can be compared to the objective function value of the incumbent solution. The resulting MIP Gap, expressed as a percentage of the incumbent solution, serves as a measure of progress toward finding and proving optimality. When active nodes no longer exist, then these two values will have converged toward each other, and the MIP Gap will thus be zero, signifying that optimality of the incumbent has been proven.
It is possible to tell IloCplex to terminate the branch & cut procedure sooner than a completed proof of optimality. For example, a user can set a time limit or a limit on the number of nodes to be processed. Indeed, with default settings, IloCplex will terminate the search when the MIP Gap has been brought lower than 0.0001 (0.01%), because it is often the case that much computation is invested in moving the Best Node value after the eventual optimal incumbent has been located. This termination criterion for the MIP Gap can be changed by the user, of course.

**Goals**

Goals can be used to control the branch and cut search in IloCplex. Goals are implemented in the class IloCplex::GoalI. The class IloCplex::Goal is the handle class. That is, it contains a pointer to an instance of IloCplex::GoalI along with accessors of objects in the implementation class.

To implement your own goal, you need to subclass IloCplex::GoalI and implement its virtual member functions execute and duplicateGoal. The method execute controls the branch & cut search. The method duplicateGoal creates a copy of the invoking goal object to be used for parallel branch & cut search. Implementing your goal can be greatly simplified if you use one of the macros ILOCPLEXGOALn.

Every branch & cut node maintains a goal stack, possibly containing IloCplex::GoalI objects. After IloCplex solves the relaxation of a node, it pops the top goal from the goal stack and calls its method execute. There are several types of goals:

- **OrGoal** is executed, IloCplex will create child nodes. Each of the child nodes will be initialized with a copy of the goal stack of the current node. Then, for each child node, the specified goal in the OrGoal is pushed onto the corresponding goal stack of the child node. Finally, the current node is deleted. (See IloCplex::GoalI::OrGoal for a more detailed discussion.)

- **If a cut goal is executed**, the constraint will be added to the current node relaxation. Constraint goals are provided to represent both local and global cuts. Local cut goals are conventionally used to express branching.

- **If AndGoal is executed**, its subgoals are simply pushed onto the stack. (See IloCplex::GoalI::AndGoal for a more detailed discussion.)

- **If IloCplex::GoalI::FailGoal is executed**, IloCplex will prune the current node; that is, it will discontinue the search at the current node. IloCplex will continue with another node if there is one still available in the tree.

- **If IloCplex::GoalI::SolutionGoal is executed**, IloCplex will attempt to inject a user-provided solution as a new incumbent. Before ILOG CPLEX
accepts the injected solution, it first tests whether the injected solution is feasible with respect to the model and goals.

◆ When ILOG CPLEX executes any other goal, the returned goal is simply pushed onto the stack.

IloCplex continues popping goals from the goal stack until OrGoal or FailGoal is executed, or until the stack becomes empty; in the case of an empty stack, it will continue with a built-in search strategy.

The predefined goals OrGoal and AndGoal allow you to combine goals. AndGoal allows you to execute different goals at one node, while OrGoal allows you to execute different goals on different, newly created nodes. A conventional use of these two goals in a return statement of a user-written goal looks like this:

```cpp
return AndGoal ( OrGoal (branch1, branch2), this);
```

This AndGoal first pushes this (the goal currently being executed) onto the goal stack, and then it pushes the OrGoal. Thus the OrGoal is on top of the stack and will be executed next. When the OrGoal executes, it creates two new nodes and copies the remaining goal stack to both of them. Thus both new nodes will have this goal on top of the goal stack at this point. Then the OrGoal proceeds to push branch1 onto the goal stack of the first child node and branch2 onto the goal stack of the second goal child node. Conventionally, branch1 and branch2 contain cut goals, so by executing branch1 and branch2 at the respective child nodes, the child nodes will be restricted to represent smaller subproblems than their parent. After branch1 and branch2 have been executed, this is on top of the node stack of both child nodes; that is, both child nodes will continue branching according to the same rule. In summary, this example creates the branches branch1 and branch2 and continues in both branches to control the same search strategy this.

To perform a search using a goal, you need to solve the extracted model by calling the method `IloCplex::solve(goal)` with the goal to use as an argument instead of the standard `IloCplex::solve`. The method `solve(goal)` simply pushes the goal onto the goal stack of the root node before calling the standard `solve`.

See Also

`IloCplex::Goal` and `IloCplex::GoalI`
The API of ILOG CPLEX for users of C++.

<table>
<thead>
<tr>
<th>Classes Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>IloCplex</td>
</tr>
<tr>
<td>IloCplex::BarrierCallbackI</td>
</tr>
<tr>
<td>IloCplex::BranchCallbackI</td>
</tr>
<tr>
<td>IloCplex::Callback</td>
</tr>
<tr>
<td>IloCplex::CallbackI</td>
</tr>
<tr>
<td>IloCplex::ContinuousCallbackI</td>
</tr>
<tr>
<td>IloCplex::ControlCallbackI</td>
</tr>
<tr>
<td>IloCplex::CrossoverCallbackI</td>
</tr>
<tr>
<td>IloCplex::CutCallbackI</td>
</tr>
<tr>
<td>IloCplex::DisjunctiveCutCallbackI</td>
</tr>
<tr>
<td>IloCplex::Exception</td>
</tr>
<tr>
<td>IloCplex::FlowMIRCutCallbackI</td>
</tr>
<tr>
<td>IloCplex::FractionalCutCallbackI</td>
</tr>
<tr>
<td>IloCplex::Goal</td>
</tr>
<tr>
<td>IloCplex::GoalI</td>
</tr>
<tr>
<td>IloCplex::HeuristicCallbackI</td>
</tr>
<tr>
<td>IloCplex::IncumbentCallbackI</td>
</tr>
<tr>
<td>IloCplex::InvalidCutException</td>
</tr>
<tr>
<td>IloCplex::LazyConstraintCallbackI</td>
</tr>
<tr>
<td>IloCplex::MIPCallbackI</td>
</tr>
<tr>
<td>IloCplex::MultipleConversionException</td>
</tr>
<tr>
<td>IloCplex::MultipleObjException</td>
</tr>
<tr>
<td>IloCplex::NetworkCallbackI</td>
</tr>
<tr>
<td>IloCplex::NodeCallbackI</td>
</tr>
<tr>
<td>IloCplex::NodeEvaluator</td>
</tr>
<tr>
<td>IloCplex::NodeEvaluatorI</td>
</tr>
<tr>
<td>IloCplex::PresolveCallbackI</td>
</tr>
<tr>
<td>IloCplex::ProbingCallbackI</td>
</tr>
<tr>
<td>IloCplex::SearchLimit</td>
</tr>
</tbody>
</table>
### IloCplex::SearchLimit

- IloCplex::SearchLimit
- IloCplex::SimplexCallback
- IloCplex::SolveCallback
- IloCplex::UnknownExtractable
- Exception
- IloCplex::UserCutCallback
- MIPCallback::NodeData

<table>
<thead>
<tr>
<th>Macros Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>ILBARIERCALLBACK0</td>
</tr>
<tr>
<td>ILBRANCHCALLBACK0</td>
</tr>
<tr>
<td>ILCONTINUOUSCALLBACK0</td>
</tr>
<tr>
<td>ILOCLEXGOAL0</td>
</tr>
<tr>
<td>ILOCROSSOVERCALLBACK0</td>
</tr>
<tr>
<td>ILOCUTCALLBACK0</td>
</tr>
<tr>
<td>ILODISJUNCTIVECUTCALLBACK0</td>
</tr>
<tr>
<td>ILOFACTIONALCUTCALLBACK0</td>
</tr>
<tr>
<td>ILOHISTICCALLBACK0</td>
</tr>
<tr>
<td>IILINCOMPDECCALLBACK0</td>
</tr>
<tr>
<td>ILOLZCONSTRANCCALLBACK0</td>
</tr>
<tr>
<td>ILOMIPCALLBACK0</td>
</tr>
<tr>
<td>ILONETWORKCALLBACK0</td>
</tr>
<tr>
<td>ILONODECALLBACK0</td>
</tr>
<tr>
<td>ILORESOLECALLBACK0</td>
</tr>
<tr>
<td>ILORINGECALLBACK0</td>
</tr>
<tr>
<td>ILSIMPLEXCALLBACK0</td>
</tr>
<tr>
<td>ILSOLVECALLBACK0</td>
</tr>
<tr>
<td>ILOUSERCUTCALLBACK0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Enumerations Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>BranchCallback::BranchType</td>
</tr>
<tr>
<td>Callback::Type</td>
</tr>
<tr>
<td>ControlCallback::IntegerFeasibility</td>
</tr>
<tr>
<td>GoalI::BranchType</td>
</tr>
<tr>
<td>GoalI::IntegerFeasibility</td>
</tr>
</tbody>
</table>
Description

IloCplex is a Concert Technology class derived from IloAlgorithm. Instances of this class are capable of solving optimization problems of the following types:

- Linear Programs (LPs),
- Mixed Integer Linear Programs (MILPs),
- Mixed Integer Programs (MIPs),
- Quadratic Programs (QPs),
- Mixed Integer Quadratic Programs (MIQPs),
- Quadratically Constrained Programs (QCPs);
Mixed Integer Quadratically Constrained Programs (MIQCPs).

An instance of IloCplex can extract and solve models consisting of the following Concert Technology extractables:

<table>
<thead>
<tr>
<th>Extractable Class</th>
<th>Used to Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>IloNumVar</td>
<td>numeric variables</td>
</tr>
<tr>
<td>IloSemiContVar</td>
<td>semi-continuous or semi-integer variables</td>
</tr>
<tr>
<td>IloObjective</td>
<td>at most one objective function with linear, piecewise linear, or quadratic expressions</td>
</tr>
<tr>
<td>IloRange</td>
<td>range constraints with linear or piecewise linear expressions</td>
</tr>
<tr>
<td>IloConstraint</td>
<td>ranged constraints of the form expr1relationexpr, where expr1 indicates a linear, logical, or quadratic expression and the relation less than or equal to or the relation greater than or equal to; constraints can be combined by logical operators</td>
</tr>
<tr>
<td>IloConversion</td>
<td>variable type conversions</td>
</tr>
<tr>
<td>IloModel</td>
<td>submodels</td>
</tr>
<tr>
<td>IloSOS1</td>
<td>special ordered sets of type 1</td>
</tr>
<tr>
<td>IloSOS2</td>
<td>special ordered sets of type 2</td>
</tr>
<tr>
<td>IloAnd</td>
<td>constraint clauses</td>
</tr>
</tbody>
</table>

What is special about this set of extractable classes recognized by IloCplex is that models consisting of these objects can be transformed into mathematical programming problems of the form:

\[
\begin{align*}
\text{min/max } & = \frac{1}{2}x^tQx \\
\text{ s.t. } & \quad L \leq Ax \leq U \\
& \quad 1 \leq x \leq u
\end{align*}
\]

When all variables are continuous and Q is zero, problems of this form are known as Linear Programs (LPs). If Q is not zero, such problems are known as Quadratic Programs (QPs). If any variables are integer, semi-continuous, or Boolean, such problems are called Mixed Integer Programs (MIPs). A MIP with a zero Q matrix is called a Mixed Integer Linear Program (MILP), and a MIP with a non-zero Q is called a Mixed Integer Quadratic Program (MIQP). If there are quadratic constraints in the problem, and its variables are continuous, it is known as a Quadratically Constrained
Program (QCP). If in addition to the quadratic constraints, there are discrete variables in the problem (such as integer, Boolean, or semi-continuous variables), then it is known as MIQCP.

- Objects of the class `IloNumVar` represent modeling variables. They are defined by the lower and upper bounds of the variable, and the type of the variable. The type of the variable can be one of these:
  - `ILOFLOAT`, for continuous,
  - `ILOINT`, for integer,
  - `ILOBOOL`, for Boolean variables.

- Objects of the class `IloSemiContVar` represent semi-continuous variables. A semi-continuous variable may be 0 (zero) or may take a value within an interval defined by its semi-continuous lower and upper bounds. Semi-continuous variables are usually defined as continuous variables, but you can designate an instance of `IloSemiContVar` as integer by using the type indicator it inherits from `IloNumVar`.

- Objects of the class `IloObjective` represent objective functions of optimization models. `IloCplex` deals with models containing at most one objective function, and the objective function must be linear, piecewise linear, or quadratic.

- Objects of the class `IloRange` represent constraints of the form: lower bound \( \leq \) expression \( \leq \) upper bound. Any floating-point value or +/- `IloInfinity` can be used for the bounds.

- Objects of the class `IloConversion` change the type of a variable in a model. This class allows you to use the same variable with different types in different models.

- Objects of the class `IloModel` represent models which consist of extractable objects. They can be used to create submodels or additional models in a given environment.

- Objects of the class `IloSOS1` represent type 1 Special Ordered Sets (SOSs). A type 1 SOS specifies that at most one variable from a set of variables may take a nonzero value. Similarly, objects of the class `IloSOS2` represent type 2 SOSs. A type 2 SOS specifies that at most two variables from a set of variables may take nonzero values and that these two variables must be neighbors with respect to a specified order of the variables. SOS1 are rarely used and SOS2 are mostly used to model piecewise linear functions, for which Concert Technology provides direct support (with the class `IloPiecewiseLinear`).

- Objects of the class `IloAnd` are used in conjunction with objects of the class `IloSolution`. 
Refer to the *ILOG Concert Technology Reference Manual* for more information about these classes.

**IloCplex Optimizer Options**

An instance of the class *IloCplex* is not really only one algorithm, but, in fact, consists of a set of highly configurable algorithms, also known as optimizer options. They include primal and dual simplex algorithms, barrier algorithm, a sifting algorithm, a network simplex algorithm, and a branch & cut algorithm for MIPs. Though in most cases *IloCplex* can be used like a black box, the optimizer options can be selected individually to provide a wealth of parameters that allow you to fine tune the algorithm to your particular model. In the case of the mixed integer optimizer, you can use your own goals or callbacks and directly control the branch & cut search carried out by *IloCplex*.

The most general kind of problem is a MIP. You might think of the LPs as a subset of MIPs: an LP is a problem in which the model is:

- without integer variables,
- without Boolean variables,
- without semi-continuous variables,
- without piecewise linear functions,
- without a quadratic component in the objective function,
- without quadratic constraints,
- and without a special ordered set (SOS).

For linear programming problems (LPs), a variety of additional solution information can be queried. These queries include dual information or, with the appropriate optimizer option, basis information. Sensitivity analysis allows you to analyze how you can modify your model while preserving the same solution. Or, if your model is infeasible, the infeasibility finder enables you to analyze the source of the infeasibility.
ILOBARRIERCALLBACK0

Category Macro

Synopsis

ILOBARRIERCALLBACK0(name)
ILOBARRIERCALLBACK1(name, type1, x1)
ILOBARRIERCALLBACK2(name, type1, x1, type2, x2)
ILOBARRIERCALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILOBARRIERCALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)
ILOBARRIERCALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5)
ILOBARRIERCALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5, type6, x6)
ILOBARRIERCALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5, type6, x6, type7, x7)

Description

This macro creates two things: an implementation class for a user-defined callback named name and a function named name that creates an instance of this class and returns a handle for it, that is, an instance of IloCplex::Callback. This function needs to be called with an environment as its first parameter, followed by the n parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the use method of an IloCplex object.

The class name that is created by the macro includes the implementation of method duplicateCallback as required for callbacks. The implementation of the main method must be provided by the user in curly brackets {} following the macro invocation, like this:

ILOBARRIERCALLBACKn(name, ...) {
// implementation of the callback
}

For the implementation of the callback, methods from class IloCplex::BarrierCallbackI and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument name is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also IloCplex::BarrierCallbackI
**ILOBRANCHCALLBACK0**

**Category**  
Macro

**Synopsis**  

- `ILOBRANCHCALLBACK0(name)`
- `ILOBRANCHCALLBACK1(name, type1, x1)`
- `ILOBRANCHCALLBACK2(name, type1, x1, type2, x2)`
- `ILOBRANCHCALLBACK3(name, type1, x1, type2, x2, type3, x3)`
- `ILOBRANCHCALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)`
- `ILOBRANCHCALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5)`
- `ILOBRANCHCALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5, type6, x6)`
- `ILOBRANCHCALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5, type6, x6, type7, x7)`

**Description**  
This macro creates two things: an implementation class for a user-defined callback named `nameI` and a function named `name` that creates an instance of this class and returns a handle for it, that is, an instance of `IloCplex::Callback`. This function needs to be called with an environment as its first parameter, followed by the `n` parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the `use` method of an `IloCplex` object.

The class `nameI` that is created by the macro includes the implementation of the method `duplicateCallback` as required for callbacks. The implementation of the main method must be provided by the user in curly brackets following the macro invocation, like this:

```
ILOBRANCHCALLBACKn{name, ...} {
    // implementation of the callback
}
```

For the implementation of the callback, methods from class `IloCplex::BranchCallbackI` and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument `name` is used to name the callback class, it is not possible to use the same name for several callback definitions.

**See Also**  

`IloCplex::BranchCallbackI`
ILOCONTINUOUSCALLBACK0

Category  
Macro

Synopsis

ILOCONTINUOUSCALLBACK0(name)
ILOCONTINUOUSCALLBACK1(name, type1, x1)
ILOCONTINUOUSCALLBACK2(name, type1, x1, type2, x2)
ILOCONTINUOUSCALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILOCONTINUOUSCALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)
ILOCONTINUOUSCALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5)
ILOCONTINUOUSCALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5, type6, x6)
ILOCONTINUOUSCALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5, type6, x6, type7, x7)

Description

This macro creates two things: an implementation class for a user-defined callback named nameI and a function named name that creates an instance of this class and returns a handle for it, that is, an instance of IloCplex::Callback. This function needs to be called with an environment as its first parameter followed by the n parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the use method of an IloCplex object.

The class nameI that is created by the macro includes the implementation of method duplicateCallback as required for callbacks. The implementation of the main method must be provided by the user in curly brackets following the macro invocation, like this:

ILOCONTINUOUSCALLBACKn(name, ...) {
    // implementation of the callback
}

For the implementation of the callback, methods from class IloCplex::ContinuousCallbackI and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument name is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also

IloCplex::ContinuousCallbackI
**ILOCPLEXGOAL0**

**Category**
Macro

**Synopsis**

- ILOCPLEXGOAL0(name)
- ILOCPLEXGOAL1(name, type0, var0)
- ILOCPLEXGOAL2(name, type0, var0, type1, var1)
- ILOCPLEXGOAL3(name, type0, var0, type1, var1, type2, var2)
- ILOCPLEXGOAL4(name, t0, v0, t1, v1, t2, v2, t3, v3)
- ILOCPLEXGOAL5(name, t0, v0, t1, v1, t2, v2, t3, v3, t4, v4)
- ILOCPLEXGOAL6(name, t0, v0, t1, v1, t2, v2, t3, v3, t4, v4, t5, v5)

**Description**

This macro defines a user goal class named `nameI` and a constructor named `name` with `n` data members, where `n` is the number following ILOCPLEXGOAL. The first parameter of this macro is always the name of the constructor to be created. What follows are `n` pairs of parameters, each parameter specifying a data member of the goal. The first parameter of such a pair specifies the type of the data member and is denoted as `Ti` in the macro definition above. The second parameter of such a pair, denoted by `datai`, specifies the data member’s name.

The constructor `name` created by this function will have `IloEnv env` as its first argument, followed by `n` additional arguments. The constructor creates a new instance of the user-written goal class `nameI` and populates its data members with the arguments following `IloEnv env` in the function argument list. The constructor `name` is what you should use to create new goal objects.

The call to the macro must be followed immediately by the `execute` method of the goal class. This method must be enclosed in curly brackets, as shown in the examples that follow. The macro will also generate an implementation of the method `duplicateGoal` that simply calls the default constructor for the new class `nameI`.

You are not obliged to use this macro to define goals. In particular, if your data members do not permit the use of the default constructor as an implementation of the method `duplicateGoal` or the default destructor, you must subclass `IloCplex::GoalI` directly and implement those methods appropriately.

Since the argument `name` is used to construct the name of the goal’s implementation class, it is not possible to use the same name for several goal definitions.

**Example**

Here’s how to define a goal with one data member:

```cpp
ILOCPLEXGOAL1(PrintX, IloInt, x) {
    IloEnv env = getEnv();
    env.out() << "PrintX: a goal with one data member" << endl;
    env.out() << x << endl;
}  
```
This macro generates code similar to the following lines:

```cpp
class PrintXI : public IloCplex::GoalI {
public:
    IloInt x;
    PrintXI(IloEnv env, IloInt arg1)
    IloCplex::Goal execute();
    IloCplex::Goal duplicateGoal();
}

PrintXI::PrintXI(IloEnv env, IloInt arg1) :
    IloCplex::GoalI(env),
    x(arg1) {
}

IloCplex::Goal PrintXI(IloEnv env, IloInt x) {
    return new PrintXI(env, x);
}

IloCplex::Goal PrintXI::execute() {
    IloEnv env = getEnv();
    env.out() << "PrintX: a goal with one data member" <<
    endl;
    env.out() << x << endl;
    return 0;
}

IloCplex::Goal PrintXI::duplicateGoal() {
    return new PrintXI(getEnv(), x);
}
```
ILOCROSSOVERCALLBACK0

Category: Macro

Synopsis:

ILOCROSSOVERCALLBACK0 (name)
ILOCROSSOVERCALLBACK1 (name, type1, x1)
ILOCROSSOVERCALLBACK2 (name, type1, x1, type2, x2)
ILOCROSSOVERCALLBACK3 (name, type1, x1, type2, x2, type3, x3)
ILOCROSSOVERCALLBACK4 (name, type1, x1, type2, x2, type3, x3, type4, x4)
ILOCROSSOVERCALLBACK5 (name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5)
ILOCROSSOVERCALLBACK6 (name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5, type6, x6)
ILOCROSSOVERCALLBACK7 (name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5, type6, x6, type7, x7)

Description:

This macro creates two things: an implementation class for a user-defined callback named name and a function named name that creates an instance of this class and returns a handle for it, that is, an instance of IloCplex::Callback. This function needs to be called with an environment as its first parameter, followed by the n parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the use method of an IloCplex object.

The class name that is created by the macro includes the implementation of the method duplicateCallback as required for callbacks. The implementation of the main method must be provided by the user in curly brackets {} following the macro invocation, like this:

ILOCROSSOVERCALLBACKn (name, ...) {
// implementation of the callback
}

For the implementation of the callback, methods from class IloCplex::CrossoverCallbackI and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument name is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also:
IloCplex::CallbackI, IloCplex::CrossoverCallbackI
## ILOCUTCALLBACK0

### Category
Macro

### Synopsis

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ILOCUTCALLBACK0(name)</td>
<td>Creates two things: an implementation class for a user-defined callback</td>
</tr>
<tr>
<td>ILOCUTCALLBACK1(name, type1, x1)</td>
<td>named name and a function named name that creates an instance of this class</td>
</tr>
<tr>
<td>ILOCUTCALLBACK2(name, type1, x1, type2, x2)</td>
<td>and a function named name that creates an instance of this class and</td>
</tr>
<tr>
<td>ILOCUTCALLBACK3(name, type1, x1, type2, x2, type3, x3)</td>
<td>returns a handle to it, that is, an instance of IloCplex::Callback. This</td>
</tr>
<tr>
<td>ILOCUTCALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)</td>
<td>function needs to be called with an environment as its first parameter,</td>
</tr>
<tr>
<td>ILOCUTCALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5)</td>
<td>followed by the ( n ) parameters specified at the macro execution in order</td>
</tr>
<tr>
<td>ILOCUTCALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5, type6, x6)</td>
<td>to create a callback. You can then use the callback by passing it to the</td>
</tr>
<tr>
<td>ILOCUTCALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5, type6, x6, type7, x7)</td>
<td>use method of an IloCplex object. The class name that is created by the</td>
</tr>
</tbody>
</table>

### Description
This macro creates two things: an implementation class for a user-defined callback named name and a function named name that creates an instance of this class and returns a handle to it, that is, an instance of IloCplex::Callback. This function needs to be called with an environment as its first parameter, followed by the \( n \) parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the use method of an IloCplex object.

The class name that is created by the macro includes the implementation of the method duplicateCallback as required for callbacks. The implementation of the main method must be provided by the user in curly brackets following the macro invocation, like this:

```cpp
ILOCUTCALLBACKn(name, ...) {
    // implementation of the callback
}
```

For the implementation of the callback, methods from class IloCplex::CutCallbackI and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument name is used to name the callback class, it is not possible to use the same name for several callback definitions.

### See Also
IloCplex::CutCallbackI
**ILODISJUNCTIVECUTCALLBACK0**

**Category**  
Macro

**Synopsis**

- `ILODISJUNCTIVECUTCALLBACK0(name)`
- `ILODISJUNCTIVECUTCALLBACK1(name, type1, x1)`
- `ILODISJUNCTIVECUTCALLBACK2(name, type1, x1, type2, x2)`
- `ILODISJUNCTIVECUTCALLBACK3(name, type1, x1, type2, x2, type3, x3)`
- `ILODISJUNCTIVECUTCALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)`
- `ILODISJUNCTIVECUTCALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5)`
- `ILODISJUNCTIVECUTCALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5, type6, x6)`
- `ILODISJUNCTIVECUTCALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5, type6, x6, type7, x7)`

**Description**

This macro creates two things: an implementation class for a user-defined callback named `nameI` and a function named `name` that creates an instance of this class and returns a handle to it, that is, an instance of `IloCplex::Callback`. This function needs to be called with an environment as its first parameter, followed by the `n` parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the `use` method of an `IloCplex` object.

The class `nameI` that is created by the macro includes the implementation of the method `duplicateCallback` as required for callbacks. The implementation of the `main` method must be provided by the user in curly brackets `{}` following the macro invocation, like this:

```
ILODISJUNCTIVECUTCALLBACKn(name, ...) {  
    // implementation of the callback  
}  
```

For the implementation of the callback, methods from class `IloCplex::DisjunctiveCutCallbackI` and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument name is used to name the callback class, it is not possible to use the same name for several callback definitions.

**See Also**

- `IloCplex::DisjunctiveCutCallbackI`
ILOFRACTIONALCUTCALLBACK0

Category       Macro

Synopsis

ILOFRACTIONALCUTCALLBACK0(name)
ILOFRACTIONALCUTCALLBACK1(name, type1, x1)
ILOFRACTIONALCUTCALLBACK2(name, type1, x1, type2, x2)
ILOFRACTIONALCUTCALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILOFRACTIONALCUTCALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)
ILOFRACTIONALCUTCALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5)
ILOFRACTIONALCUTCALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5, type6, x6)
ILOFRACTIONALCUTCALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5, type6, x6, type7, x7)

Description

This macro creates two things: an implementation class for a user-defined callback named nameI and a function named name that creates an instance of this class and returns a handle for it, that is, an instance of IloCplex::Callback. This function needs to be called with an environment as its first parameter, followed by the n parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the use method of an IloCplex object.

The class nameI that is created by the macro includes the implementation of the method duplicateCallback as required for callbacks. The implementation of the main method must be provided by the user in curly brackets {} following the macro invocation, like this:

ILOFRACTIONALCUTCALLBACKn(name, ...) {
  // implementation of the callback
}

For the implementation of the callback, methods from class IloCplex::FractionalCutCallbackI and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument name is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also

IloCplex::FractionalCutCallbackI
ILOHEURISTICCALLBACK0

Category  Macro

Synopsis  
ILOHEURISTICCALLBACK0(name)
ILOHEURISTICCALLBACK1(name, type1, x1)
ILOHEURISTICCALLBACK2(name, type1, x1, type2, x2)
ILOHEURISTICCALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILOHEURISTICCALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)
ILOHEURISTICCALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5)
ILOHEURISTICCALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5, type6, x6)
ILOHEURISTICCALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5, type6, x6, type7, x7)

Description  
This macro creates two things: an implementation class for a user-defined callback named `name` and a function named `name` that creates an instance of this class and returns a handle for it, that is, an instance of `IloCplex::Callback`. This function needs to be called with an environment as its first parameter, followed by the n parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the `use` method of an `IloCplex` object.

The class `nameI` that is created by the macro includes the implementation of the method `duplicateCallback` as required for callbacks. The implementation of the `main` method must be provided by the user in curly brackets following the macro invocation, like this:

```
ILOHEURISTICCALLBACKn(name, ...) {
    // implementation of the callback
}
```

For the implementation of the callback, methods from class `IloCplex::HeuristicCallbackI` and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument `name` is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also  
`IloCplex::HeuristicCallbackI`
ILOINCUMBENTCALLBACK0

Category
Macro

Synopsis

ILOINCUMBENTCALLBACK0(name)
ILOINCUMBENTCALLBACK1(name, type1, x1)
ILOINCUMBENTCALLBACK2(name, type1, x1, type2, x2)
ILOINCUMBENTCALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILOINCUMBENTCALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)
ILOINCUMBENTCALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4,
 type5, x5)
ILOINCUMBENTCALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4,
 type5, x5, type6, x6)
ILOINCUMBENTCALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4,
 type5, x5, type6, x6, type7, x7)

Description
This macro creates two things: an implementation class for a user-defined callback
named nameI and a function named name that creates an instance of this class and
returns a handle for it, that is, an instance of IloCplex::Callback. This function
needs to be called with an environment as its first parameter, followed by the n
parameters specified at the macro execution in order to create a callback. You can then
use the callback by passing it to the use method of an IloCplex object.

The class nameI that is created by the macro includes the implementation of the
method duplicateCallback as required for callbacks. The implementation of the
main method must be provided by the user in curly brackets following the macro
invocation, like this:

ILOINCUMBENTCALLBACKn(name, ...) {
    // implementation of the callback
}

For the implementation of the callback, methods from class
IloCplex::IncumbentCallbackI and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too
restrictive for your purposes, we recommend that you define a callback class directly.
Since the argument name is used to name the callback class, it is not possible to use the
same name for several callback definitions.

See Also
IloCplex::IncumbentCallbackI
**ILOLAZYCONSTRAINTCALLBACK0**

**Category**  
Macro

**Synopsis**

- `ILOLAZYCONSTRAINTCALLBACK0(name)`
- `ILOLAZYCONSTRAINTCALLBACK1(name, type1, x1)`
- `ILOLAZYCONSTRAINTCALLBACK2(name, type1, x1, type2, x2)`
- `ILOLAZYCONSTRAINTCALLBACK3(name, type1, x1, type2, x2, type3, x3)`
- `ILOLAZYCONSTRAINTCALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)`
- `ILOLAZYCONSTRAINTCALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5)`
- `ILOLAZYCONSTRAINTCALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5, type6, x6)`
- `ILOLAZYCONSTRAINTCALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5, type6, x6, type7, x7)`

**Description**

This macro creates two things: an implementation class for a user-defined lazy constraint callback named `nameI` and a function named `name` that creates an instance of this class and returns an `IloCplex::Callback` handle for it. This function needs to be called with an environment as first parameter followed by the `n` parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the `use` method of an `IloCplex` object.

The class `nameI` that is created by the macro includes the implementation of method `makeClone` as required for callbacks. The implementation of the `main` method must be provided by the user in curly brackets `{}` following the macro invocation, like this:

```cpp
ILOLAZYCONSTRAINTCALLBACKn(name, ...) {
    // implementation of the callback
}
```

For the implementation of the callback, methods from class `IloCplex::LazyConstraintCallbackI` and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly.

Since the argument `name` is used to name the callback class, it is not possible to use the same name for several callback definitions.

**See Also**

- `IloCplex::LazyConstraintCallbackI`
ILOMIPCALLBACK0

Category
Macro

Synopsis
ILOMIPCALLBACK0(name)
ILOMIPCALLBACK1(name, type1, x1)
ILOMIPCALLBACK2(name, type1, x1, type2, x2)
ILOMIPCALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILOMIPCALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)
ILOMIPCALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5)
ILOMIPCALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5, type6, x6)
ILOMIPCALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5, type6, x6, type7, x7)

Description
This macro creates two things: an implementation class for a user-defined callback named nameI and a function named name that creates an instance of this class and returns a handle for it, that is, an instance of IloCplex::Callback. This function needs to be called with an environment as its first parameter, followed by the n parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the use method of an IloCplex object.

The class nameI that is created by the macro includes the implementation of the method duplicateCallback as required for callbacks. The implementation of the main method must be provided by the user in curly brackets {} following the macro invocation, like this:

ILOMIPCALLBACKn(name, ...) {
    // implementation of the callback
}

For the implementation of the callback, methods from class IloCplex::MIPCallbackI and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument name is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also
IloCplex::MIPCallbackI
ILONETWORKCALLBACK0

Category          Macro

Synopsis
ILONETWORKCALLBACK0(name)
ILONETWORKCALLBACK1(name, type1, x1)
ILONETWORKCALLBACK2(name, type1, x1, type2, x2)
ILONETWORKCALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILONETWORKCALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)
ILONETWORKCALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5)
ILONETWORKCALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5, type6, x6)
ILONETWORKCALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5, type6, x6, type7, x7)

Description
This macro creates two things: an implementation class for a user-defined callback named nameI and a function named name that creates an instance of this class and returns a handle for it, that is, an instance of IloCplex::Callback. This function needs to be called with an environment as its first parameter, followed by the n parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the use method of an IloCplex object.

The class nameI that is created by the macro includes the implementation of the method duplicateCallback as required for callbacks. The implementation of the main method must be provided in curly brackets {} by the user following the macro invocation, like this:

ILONETWORKCALLBACKn(name, ...) {
  // implementation of the callback
}

For the implementation of the callback, methods from class IloCplex::NetworkCallbackI and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument name is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also
IloCplex::NetworkCallbackI
ILONODECALLBACK0

Category  
Macro

Synopsis  
ILONODECALLBACK0(name)
ILONODECALLBACK1(name, type1, x1)
ILONODECALLBACK2(name, type1, x1, type2, x2)
ILONODECALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILONODECALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)
ILONODECALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5)
ILONODECALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5, type6, x6)
ILONODECALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5, type6, x6, type7, x7)

Description  
This macro creates two things: an implementation class for a user-defined callback named nameI and a function named name that creates an instance of this class and returns a handle for it, that is, an instance of IloCplex::Callback. This function needs to be called with an environment as its first parameter, followed by the n parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the use method of an IloCplex object.

The class nameI that is created by the macro includes the implementation of the method duplicateCallback as required for callbacks. The implementation of the main method must be provided by the user in curly brackets following the macro invocation, like this:

ILONODECALLBACKn(name, ...) {
  // implementation of the callback
}

For the implementation of the callback, methods from class IloCplex::NodeCallbackI and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument name is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also  
IloCplex::NodeCallbackI
**ILOPRESOLVECALLBACK0**

**Category**  
Macro

**Synopsis**  
ILOPRESOLVECALLBACK0(name)  
ILOPRESOLVECALLBACK1(name, type1, x1)  
ILOPRESOLVECALLBACK2(name, type1, x1, type2, x2)  
ILOPRESOLVECALLBACK3(name, type1, x1, type2, x2, type3, x3)  
ILOPRESOLVECALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)  
ILOPRESOLVECALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5)  
ILOPRESOLVECALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5, type6, x6)  
ILOPRESOLVECALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5, type6, x6, type7, x7)

**Description**  
This macro creates two things: an implementation class for a user-defined callback named nameI and a function named name that creates an instance of this class and returns a handle for it, that is, an instance of IloCplex::Callback. This function needs to be called with an environment as its first parameter, followed by the n parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the use method of an IloCplex object.

The class nameI that is created by the macro includes the implementation of the method duplicateCallback as required for callbacks. The implementation of the main method must be provided by the user in curly brackets following the macro invocation, like this:

```
ILOPRESOLVECALLBACKn(name, ...) {
    // implementation of the callback
}
```

For the implementation of the callback, methods from class IloCplex::PresolveCallbackI and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument name is used to name the callback class, it is not possible to use the same name for several callback definitions.

**See Also**  
IloCplex::PresolveCallbackI
ILOPROBINGCALLBACK0

Category
Macro

Synopsis
ILOPROBINGCALLBACK0(name)
ILOPROBINGCALLBACK1(name, type1, x1)
ILOPROBINGCALLBACK2(name, type1, x1, type2, x2)
ILOPROBINGCALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILOPROBINGCALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)
ILOPROBINGCALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5)
ILOPROBINGCALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5, type6, x6)
ILOPROBINGCALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5, type6, x6, type7, x7)

Description
This macro creates two things: an implementation class for a user-defined callback named nameI and a function named name that creates an instance of this class and returns a handle for it, that is, an instance of IloCplex::Callback. This function needs to be called with an environment as its first parameter, followed by the n parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the use method of an IloCplex object.

The class nameI that is created by the macro includes the implementation of the method duplicateCallback as required for callbacks. The implementation of the main method must be provided by the user in curly brackets following the macro invocation, like this:

ILOPROBINGCALLBACKn(name, ...) {
    // implementation of the callback
}

For the implementation of the callback, methods from class IloCplex::ProbingCallbackI and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument name is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also
IloCplex::ProbingCallbackI
ILOSIMPLEXCALLBACK0

Category  Macro

Synopsis  ILOSIMPLEXCALLBACK0(name)
ILOSIMPLEXCALLBACK1(name, type1, x1)
ILOSIMPLEXCALLBACK2(name, type1, x1, type2, x2)
ILOSIMPLEXCALLBACK3(name, type1, x1, type2, x2, type3, x3)
ILOSIMPLEXCALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)
ILOSIMPLEXCALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5)
ILOSIMPLEXCALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5, type6, x6)
ILOSIMPLEXCALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5, type6, x6, type7, x7)

Description  This macro creates two things: an implementation class for a user-defined callback named nameI and a function named name that creates an instance of this class and returns a handle for it, that is, an instance of IloCplex::Callback. This function needs to be called with an environment as its first parameter, followed by the n parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the use method of an IloCplex object.

The class nameI that is created by the macro includes the implementation of the method duplicateCallback as required for callbacks. The implementation of the main method must be provided by the user in curly brackets following the macro invocation, like this:

ILOSIMPLEXCALLBACKn(name, ...) {
  // implementation of the callback
}

For the implementation of the callback, methods from the class IloCplex::SimplexCallbackI and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument name is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also  IloCplex::SimplexCallbackI
ILOSOLVECALLBACK

Category
Macro

Synopsis

- ILOSOLVECALLBACK0(name)
- ILOSOLVECALLBACK1(name, type1, x1)
- ILOSOLVECALLBACK2(name, type1, x1, type2, x2)
- ILOSOLVECALLBACK3(name, type1, x1, type2, x2, type3, x3)
- ILOSOLVECALLBACK4(name, type1, x1, type2, x2, type3, x3, type4, x4)
- ILOSOLVECALLBACK5(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5)
- ILOSOLVECALLBACK6(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5, type6, x6)
- ILOSOLVECALLBACK7(name, type1, x1, type2, x2, type3, x3, type4, x4, type5, x5, type6, x6, type7, x7)

Description

This macro creates two things: an implementation class for a user-defined callback named name and a function named name that creates an instance of this class and returns a handle for it, that is, an instance of IloCplex::Callback. This function needs to be called with an environment as its first parameter, followed by the n parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the use method of an IloCplex object.

The class name that is created by the macro includes the implementation of the method duplicateCallback as required for callbacks. The implementation of the main method must be provided by the user in curly brackets following the macro invocation, like this:

ILOSOLVECALLBACKn(name, ...) {

// implementation of the callback

}

For the implementation of the callback, methods from the class IloCplex:: SolveCallbackI and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument name is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also
IloCplex:: SolveCallbackI
ILOUSERCUTCALLBACK0

Category  Macro

Synopsis  
ILOUSERCUTCALLBACK0(name)
ILOUSERCUTCALLBACK1(name, typel, x1)
ILOUSERCUTCALLBACK2(name, typel, x1, typetwo, x2)
ILOUSERCUTCALLBACK3(name, typel, x1, typetwo, x2, typethree, x3)
ILOUSERCUTCALLBACK4(name, typel, x1, typetwo, x2, typethree, x3, typefour, x4)
ILOUSERCUTCALLBACK5(name, typel, x1, typetwo, x2, typethree, x3, typefour, x4, typefive, x5)
ILOUSERCUTCALLBACK6(name, typel, x1, typetwo, x2, typethree, x3, typefour, x4, typefive, x5, typesix, x6)
ILOUSERCUTCALLBACK7(name, typel, x1, typetwo, x2, typethree, x3, typefour, x4, typefive, x5, typesix, x6, typeseven, x7)

Description  
This macro creates two things: an implementation class for a user-defined user cut callback named nameI and a function named name that creates an instance of this class and returns an IloCplex::Callback handle for it. This function needs to be called with an environment as first parameter followed by the n parameters specified at the macro execution in order to create a callback. You can then use the callback by passing it to the use method of an IloCplex object.

The class nameI that is created by the macro includes the implementation of the method makeClone as required for callbacks. The implementation of the main method must be provided by the user in curly brackets {} following the macro invocation, like this:

ILOUSERCUTCALLBACKn(name, ...) {

// implementation of the callback

}

For the implementation of the callback, methods from class IloCplex::UserCutCallbackI and its parent classes can be used.

You are not obliged to use this macro to define callbacks. When the macro seems too restrictive for your purposes, we recommend that you define a callback class directly. Since the argument name is used to name the callback class, it is not possible to use the same name for several callback definitions.

See Also  IloCplex::UserCutCallbackI
IloCplex

Category
Class

InheritancePath

Definition File
ilocplex1.h

Include Files
ilcplex/ilcplex.h

Constructor Summary

<table>
<thead>
<tr>
<th>Constructor Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>public IloCplex(IloEnv)</td>
</tr>
<tr>
<td>public IloCplex(const IloModel)</td>
</tr>
</tbody>
</table>

Method Summary

<table>
<thead>
<tr>
<th>Method Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>public IloConstraint addCut(IloConstraint)</td>
</tr>
<tr>
<td>public const IloConstraintArray addCuts(const IloConstraintArray)</td>
</tr>
<tr>
<td>public IloConstraint addLazyConstraint(IloConstraint)</td>
</tr>
<tr>
<td>public const IloConstraintArray addLazyConstraints(const IloConstraintArray)</td>
</tr>
<tr>
<td>public IloConstraint addUserCut(IloConstraint)</td>
</tr>
<tr>
<td>public const IloConstraintArray addUserCuts(const IloConstraintArray)</td>
</tr>
<tr>
<td>public static IloCplex::Goal Apply(IloCplex, IloCplex::Goal, IloCplex::NodeEvaluator)</td>
</tr>
<tr>
<td>public void basicPresolve(const IloNumVarArray, IloNumArray, IloNumArray, const IloRangeArray, IloBoolArray)</td>
</tr>
<tr>
<td>public void clearCuts()</td>
</tr>
<tr>
<td>public void clearLazyConstraints()</td>
</tr>
<tr>
<td>public void clearModel()</td>
</tr>
<tr>
<td>Method</td>
</tr>
<tr>
<td>--------------------------------</td>
</tr>
<tr>
<td>public void clearUserCuts()</td>
</tr>
<tr>
<td>public void delDirection(IloNumVar)</td>
</tr>
<tr>
<td>public void delDirections(const IloNumVarArray)</td>
</tr>
<tr>
<td>public void delPriorities(const IloNumVarArray)</td>
</tr>
<tr>
<td>public void delPriority(IloNumVar)</td>
</tr>
<tr>
<td>public void exportModel(const char *)</td>
</tr>
<tr>
<td>public IloBool feasOpt(const IloRangeArray, IloNumArray, IloNumArray, IloBool)</td>
</tr>
<tr>
<td>public IloBool feasOpt(const IloNumVarArray, IloNumArray, IloNumArray, IloBool)</td>
</tr>
<tr>
<td>public IloBool feasOpt(const IloRangeArray, IloNumArray, IloNumArray, const IloNumVarArray, IloNumArray, IloNumArray, IloBool)</td>
</tr>
<tr>
<td>public void freePresolve()</td>
</tr>
<tr>
<td>public IloCplex::Algorithm getAlgorithm()</td>
</tr>
<tr>
<td>public void getAX(IloNumArray, const IloRangeArray)</td>
</tr>
<tr>
<td>public IloNum getAX(const IloRange)</td>
</tr>
<tr>
<td>public IloCplex::BasisStatus getBasisStatus(const IloConstraint)</td>
</tr>
<tr>
<td>public IloCplex::BasisStatus getBasisStatus(const IloIntVar)</td>
</tr>
<tr>
<td>public IloCplex::BasisStatus getBasisStatus(const IloNumVar)</td>
</tr>
<tr>
<td>public void getBasisStatuses(IloCplex::BasisStatusArray, const IloNumVarArray, IloCplex::BasisStatusArray, const IloConstraintArray)</td>
</tr>
<tr>
<td>public void getBasisStatuses(IloCplex::BasisStatusArray, const IloConstraintArray)</td>
</tr>
<tr>
<td>public void getBasisStatuses(IloCplex::BasisStatusArray, const IloNumVarArray)</td>
</tr>
<tr>
<td>public IloNum getBestObjValue()</td>
</tr>
<tr>
<td>public void getBoundSA(IloNumArray, IloNumArray, IloNumArray, IloNumArray, const IloNumVarArray)</td>
</tr>
<tr>
<td>Method</td>
</tr>
<tr>
<td>---------------------------------------------</td>
</tr>
<tr>
<td><code>public IloCplex::CplexStatus getCplexStatus()</code></td>
</tr>
<tr>
<td><code>public IloCplex::CplexStatus getCplexSubStatus()</code></td>
</tr>
<tr>
<td><code>public IloNum getCutoff()</code></td>
</tr>
<tr>
<td><code>public IloBool getDefault(IloCplex::BoolParam)</code></td>
</tr>
<tr>
<td><code>public IloCplex::DeleteMode getDeleteMode()</code></td>
</tr>
<tr>
<td><code>public IloCplex::BranchDirection getDirection(IloNumVar)</code></td>
</tr>
<tr>
<td><code>public void getDirections(IloCplex::BranchDirectionArray, const IloNumVarArray)</code></td>
</tr>
<tr>
<td><code>public IloExtractable getDiverging()</code></td>
</tr>
<tr>
<td><code>public IloNum getDual(const IloRange)</code></td>
</tr>
<tr>
<td><code>public void getDuals(IloNumArray, const IloRangeArray)</code></td>
</tr>
<tr>
<td><code>public IloNum getFormulationBigValue()</code></td>
</tr>
<tr>
<td><code>public IloNum getFormulationEpsValue()</code></td>
</tr>
<tr>
<td><code>public void getIIS(IloCplex::IISStatusArray, IloNumVarArray, IloCplex::IISStatusArray, IloConstraintArray)</code></td>
</tr>
<tr>
<td><code>public IloInt getIncumbentNode()</code></td>
</tr>
<tr>
<td><code>public IloInt getMax(IloCplex::IntParam)</code></td>
</tr>
<tr>
<td><code>public IloInt getMin(IloCplex::IntParam)</code></td>
</tr>
<tr>
<td><code>public IloInt getNbarrierIterations()</code></td>
</tr>
<tr>
<td><code>public IloInt getNbinVars()</code></td>
</tr>
<tr>
<td><code>public IloInt getNcliques()</code></td>
</tr>
<tr>
<td><code>public IloInt getNcols()</code></td>
</tr>
<tr>
<td><code>public IloInt getNcovers()</code></td>
</tr>
<tr>
<td><code>public IloInt getNcrossDExch()</code></td>
</tr>
<tr>
<td><code>public IloInt getNcrossDPush()</code></td>
</tr>
<tr>
<td><code>public IloInt getNcrossPExch()</code></td>
</tr>
<tr>
<td><code>public IloInt getNcrossPPush()</code></td>
</tr>
<tr>
<td><code>public IloInt getNdualSuperbasics()</code></td>
</tr>
<tr>
<td><code>public IloInt getNintVars()</code></td>
</tr>
<tr>
<td><code>public IloInt getNiterations()</code></td>
</tr>
<tr>
<td><code>public IloInt getNnodes()</code></td>
</tr>
<tr>
<td><code>public IloInt getNnodesLeft()</code></td>
</tr>
<tr>
<td><code>public IloInt getNNZs()</code></td>
</tr>
<tr>
<td><code>public IloInt getNphaseOneIterations()</code></td>
</tr>
<tr>
<td><code>public IloInt getNprimalSuperbasics()</code></td>
</tr>
<tr>
<td><code>public IloInt getNQCs()</code></td>
</tr>
<tr>
<td><code>public IloInt getNrows()</code></td>
</tr>
<tr>
<td><code>public IloInt getNsemiContVars()</code></td>
</tr>
<tr>
<td>Method</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>public IloInt</td>
</tr>
<tr>
<td>public IloInt</td>
</tr>
<tr>
<td>public IloInt</td>
</tr>
<tr>
<td>public IloInt</td>
</tr>
<tr>
<td>public void</td>
</tr>
<tr>
<td>public IloBool</td>
</tr>
<tr>
<td>public void</td>
</tr>
<tr>
<td>public IloNum</td>
</tr>
<tr>
<td>public IloCplex::PWLFormulation</td>
</tr>
<tr>
<td>public IloNum</td>
</tr>
<tr>
<td>public void</td>
</tr>
<tr>
<td>public void</td>
</tr>
<tr>
<td>public IloNum</td>
</tr>
<tr>
<td>public IloNum</td>
</tr>
<tr>
<td>public void</td>
</tr>
<tr>
<td>public void</td>
</tr>
<tr>
<td>public void</td>
</tr>
<tr>
<td>public IloNum</td>
</tr>
<tr>
<td>public void</td>
</tr>
<tr>
<td>public IloAlgorithm::Status</td>
</tr>
<tr>
<td>public IloCplex::Algorithm</td>
</tr>
<tr>
<td>public void</td>
</tr>
<tr>
<td>public void</td>
</tr>
<tr>
<td>public void</td>
</tr>
<tr>
<td>public const char *</td>
</tr>
<tr>
<td>Method</td>
</tr>
<tr>
<td>--------------------------------</td>
</tr>
<tr>
<td>public void importModel</td>
</tr>
<tr>
<td>public void importModel</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>public void importModel</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>public IloBool isDualFeasible</td>
</tr>
<tr>
<td>public IloBool isMIP()</td>
</tr>
<tr>
<td>public IloBool isPrimalFeasible</td>
</tr>
<tr>
<td>public IloBool isQC()</td>
</tr>
<tr>
<td>public IloBool isQO()</td>
</tr>
<tr>
<td>public static IloCplex::Goal</td>
</tr>
<tr>
<td>public void presolve</td>
</tr>
<tr>
<td>public void qpIndefCertificate</td>
</tr>
<tr>
<td>public void readBasis</td>
</tr>
<tr>
<td>public void readMIPStart</td>
</tr>
<tr>
<td>public void readOrder</td>
</tr>
<tr>
<td>public void readParam</td>
</tr>
<tr>
<td>public void readTree</td>
</tr>
<tr>
<td>public void readVectors</td>
</tr>
<tr>
<td>public void setBasisStatuses</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>public void setDefaults</td>
</tr>
<tr>
<td>public void setDeleteMode</td>
</tr>
<tr>
<td>public void setDirection</td>
</tr>
</tbody>
</table>
### IloCplex

**Inner Enumeration**

- `IloCplex::Algorithm`
- `IloCplex::BasisStatus`
- `IloCplex::BoolParam`
- `IloCplex::BranchDirection`
- `IloCplex::CplexStatus`
- `IloCplex::DeleteMode`
- `IloCplex::DualPricing`
- `IloCplex::IISStatus`
- `IloCplex::IntParam`
- `IloCplex::MIFEmphasisType`

**Public Member Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public void</td>
<td>setDirections(const IloNumVarArray, const IloCplex::BranchDirectionArray)</td>
</tr>
<tr>
<td>public void</td>
<td>setFormulationBigValue(IloNum)</td>
</tr>
<tr>
<td>public void</td>
<td>setFormulationEpsValue(IloNum)</td>
</tr>
<tr>
<td>public void</td>
<td>setParam(IloCplex::BoolParam, IloBool)</td>
</tr>
<tr>
<td>public void</td>
<td>setPriorities(const IloNumVarArray, const IloNumArray)</td>
</tr>
<tr>
<td>public void</td>
<td>setPriority(IloNumVar, IloNum)</td>
</tr>
<tr>
<td>public void</td>
<td>setPWLFormulation(IloCplex::PWLFormulation)</td>
</tr>
<tr>
<td>public void</td>
<td>setVectors(const IloNumArray, const IloNumArray, const IloNumVarArray, const IloNumArray, const IloNumArray, const IloRangeArray)</td>
</tr>
<tr>
<td>public IloBool</td>
<td>solve(IloCplex::Goal)</td>
</tr>
<tr>
<td>public IloBool</td>
<td>solve()</td>
</tr>
<tr>
<td>public IloBool</td>
<td>solveFixed()</td>
</tr>
<tr>
<td>public IloCplex::Callback</td>
<td>use(IloCplex::Callback)</td>
</tr>
<tr>
<td>public void</td>
<td>writeBasis(const char *)</td>
</tr>
<tr>
<td>public void</td>
<td>writeMIPStart(const char *)</td>
</tr>
<tr>
<td>public void</td>
<td>writeOrder(const char *)</td>
</tr>
<tr>
<td>public void</td>
<td>writeParam(const char *)</td>
</tr>
<tr>
<td>public void</td>
<td>writeTextSolution(const char *)</td>
</tr>
<tr>
<td>public void</td>
<td>writeTree(const char *)</td>
</tr>
<tr>
<td>public void</td>
<td>writeVectors(const char *)</td>
</tr>
</tbody>
</table>
Description

IloCplex derives from the class IloAlgorithm. Use it to solve LP (linear programming), QP (programs with quadratic terms in the objective function), QCP (quadratically constrained programming), and MIP (mixed integer programming).

## Inner Type Def

<table>
<thead>
<tr>
<th>Type Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>IloCplex::IloCplex::BasisStatusArray</td>
</tr>
<tr>
<td>IloCplex::IloCplex::BranchDirectionArray</td>
</tr>
<tr>
<td>IloCplex::IloCplex::IISStatusArray</td>
</tr>
</tbody>
</table>

## IloCplex Exception Classes

<table>
<thead>
<tr>
<th>Exception Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>IloCplex::IloCplex::InvalidCutException</td>
</tr>
<tr>
<td>IloCplex::IloCplex::PresolveCallbackI</td>
</tr>
<tr>
<td>IloCplex::IloCplex::SimplexCallbackI</td>
</tr>
<tr>
<td>IloCplex::IloCplex::NetworkCallbackI</td>
</tr>
<tr>
<td>IloCplex::IloCplex::MIPCallbackI</td>
</tr>
<tr>
<td>IloCplex::IloCplex::ProbingCallbackI</td>
</tr>
<tr>
<td>IloCplex::IloCplex::IncumbentCallbackI</td>
</tr>
<tr>
<td>IloCplex::IloCplex::NodeCallbackI</td>
</tr>
<tr>
<td>IloCplex::IloCplex::SolveCallbackI</td>
</tr>
<tr>
<td>IloCplex::IloCplex::UserCutCallbackI</td>
</tr>
<tr>
<td>IloCplex::IloCplex::LazyConstraintCallbackI</td>
</tr>
<tr>
<td>IloCplex::IloCplex::NodeEvaluatorI</td>
</tr>
<tr>
<td>IloCplex::IloCplex::NodeEvaluator</td>
</tr>
<tr>
<td>IloCplex::IloCplex::SearchLimitI</td>
</tr>
<tr>
<td>IloCplex::IloCplex::SearchLimit</td>
</tr>
</tbody>
</table>
Mathematical Programming models. More precisely, models to be solved by IloCplex should contain only IloExtractable objects from the following list:

- **variables**: objects of type IloNumVar and its extensions IloIntVar and IloSemiContVar
- **range constraints**: objects of type IloRange
- **other relational constraints**: objects of type IloConstraint of the form $expr1$ relation $expr2$, where the relation is one of $==, \geq, \leq, $ or $!=
- **objective function**: one object of type IloObjective
- **variable type conversions**: objects of type IloConversion
- **special ordered sets**: objects of type IloSOS1 or IloSOS2

The expressions used in the constraints and objective function handled by IloCplex are built from variables of those listed types and can be linear or quadratic. In addition, expressions may contain the following constructs:

- **minimum**: IloMin
- **maximum**: IloMax
- **absolute value**: IloAbs
- **piecewise linear functions**: IloPiecewiseLinear

Expressions that evaluate only to 0 (zero) and 1 (one) are referred to as Boolean expressions. Such expressions also support:

- **negation**: operator !
- **conjunction**: operator && or, equivalently, IloAnd
- **disjunction**: operator || or, equivalently, IloOr

Moreover, Boolean expressions can be constructed not only from variables, but also from constraints.

IloCplex will automatically transform all of these constructs into an equivalent representation amenable to IloCplex. Such models can be represented in the following way:

Minimize (or Maximize) $c'x + x'Qx$
subject to $L \leq Ax \leq U$
$a_i'x + x'Q_i x \leq r_i$, for $i = 1, \ldots, q$
$L \leq x \leq u.$
That is, in fact, the standard math programming matrix representation that IloCplex uses internally. A is the matrix of linear constraint coefficients, and L and U are the vectors of lower and upper bounds on the vector of variables in the array x. The Q matrix must be positive semi-definite (or negative semi-definite in the maximization case) and represents the quadratic terms of the objective function. The matrices Q_i must be positive semi-definite and represent the quadratic terms of the i-th quadratic constraint. The a_i are vectors containing the corresponding linear terms. For details about the Q_i, see the chapter about quadratically constrained programs (QCP) in the ILOG CPLEX User's Manual.

Special ordered sets (SOS) fall outside the conventional representation in terms of A and Q matrices and are stored separately.

If the model contains integer, Boolean, or semi-continuous variables, or if the model has special ordered sets (SOSs), the model is referred to as a mixed integer program (MIP). You can query whether the active model is a MIP with the method isMIP.

A model with quadratic terms in the objective is referred to as a mixed integer quadratic program (MIQP) if it is also a MIP, and a quadratic program (QP) otherwise. You can query whether the active model has a quadratic objective by calling method isQO.

A model with quadratic constraints is referred to as a mixed integer quadratically constrained program (MIQCP) if it is also a MIP, and as a quadratically constrained program (QCP) otherwise. You can query whether the active model is quadratically constrained by calling the method isQC. A QCP may or may not have a quadratic objective; that is, a given problem may be both QP and QCP. Likewise, a MIQCP may or may not have a quadratic objective; that is, a given problem may be both MIQP and MIQCP.

If there are no quadratic terms in the objective, no integer constraints, and the problem is not quadratically constrained, and all variables are continuous it is called a linear program (LP).

Information related to the matrix representation of the model can be queried through these methods:

- `getNcols` for querying the number of columns of A,
- `getNrows` for querying the number of rows of A; that is, the number of linear constraints,
- `getNQCs` for querying the number of quadratic constraints,
- `getNNZs` for querying the number of nonzero elements in A, and
- `getNSOSs` for querying the number of special ordered sets (SOSs).

Additional information about the active model can be obtained through iterators defined on the different types of modeling objects in the extracted or active model.
IloCplex effectively treats all models as MIQCP models. That is, it allows the most general case, although the solution algorithms make efficient use of special cases, such as taking advantage of the absence of quadratic terms in the formulation. The method `solve` begins by solving the root relaxation of the MIQCP model, where all integrality constraints and SOSs are ignored. If the model has no integrality constraints or SOSs, then the optimization is complete once the root relaxation is solved. Otherwise, IloCplex uses a branch and cut procedure to reintroduce the integrality constraints or SOSs. See the ILOG CPLEX User's Manual for more information about branch & cut.

Most users can simply call `solve` to solve their models. However, several parameters are available for users who require more control. Perhaps the most important one is `IloCplex::RootAlg`, which determines the algorithm used to solve the root relaxation. Possible settings, as defined in `IloCplex::Algorithm`, are:

- **IloCplex::Auto**: IloCplex automatically selects an algorithm. This is the default setting.
- **IloCplex::Primal**: Use the primal simplex algorithm. This option is not available for quadratically constrained problems (QCPs).
- **IloCplex::Dual**: Use the dual simplex algorithm. This option is not available for quadratically constrained problems (QCPs).
- **IloCplex::Network**: Use network simplex on the embedded network part of the model, followed by dual simplex on the entire model. This option is not available for quadratically constrained problems.
- **IloCplex::Barrier**: Use the barrier algorithm.
- **IloCplex::Sifting**: Use the sifting algorithm. This option is not available for quadratic problems. If selected nonetheless, IloCplex defaults to the `IloCplex::Auto` setting.
- **IloCplex::Concurrent**: Use the several algorithms concurrently. This option is not available for quadratic problems. If selected nonetheless, IloCplex defaults to the `IloCplex::Auto` setting.

Numerous other parameters allow you to control algorithmic aspects of the optimizer. See the nested enumerations `IloCplex::IntParam`, `IloCplex::DoubleParam`, and `IloCplex::StringParam` for further information. Parameters are set with the method `setParam`.

Even higher levels of control can be achieved through goals (see `IloCplex::Goal`) or through callbacks (see `IloCplex::Callback` and its extensions).

**Information about a Solution**

The `solve` method returns an `IloBool` value indicating whether (IloTrue) or not (IloFalse) a solution (not necessarily the optimal one) has been found. Further information about the solution can be queried with the method `getStatus`. The return
code of type IloAlgorithm::Status indicates whether the solution is feasible, bounded, or optimal, or if the model has been proven to be infeasible or unbounded.

The method getCplexStatus provides more detailed information about the status of the optimizer after solve returns. For example, it can provide information on why the optimizer terminated prematurely (time limit, iteration limit, or other similar limits). The methods isPrimalFeasible and isDualFeasible determine whether a primal or dual feasible solution has been found and can be queried.

The most important solution information computed by IloCplex are usually the solution vector and the objective function value. The method IloCplex::getValue queries the solution vector. The method IloCplex::getObjValue queries the objective function value. Most optimizers also compute additional solution information, such as dual values, reduced costs, simplex bases, and others. This additional information can also be queried through various methods of IloCplex. If you attempt to retrieve solution information that is not available from a particular optimizer, IloCplex will throw an exception.

If you are solving an LP and a basis is available, the solution can be further analyzed by performing sensitivity analysis. This information tells you how sensitive the solution is with respect to changes in variable bounds, constraint bounds, or objective coefficients. The information is computed and accessed with the methods getBoundSA, getRangeSA, getRHSSA, and getObjSA.

An important consideration when you access solution information is the numeric quality of the solution. Since IloCplex performs arithmetic operations using finite precision, solutions are always subject to numeric errors. For most problems, numeric errors are well within reasonable tolerances. However, for numerically difficult models, you are advised to verify the quality of the solution using the method getQuality, which offers a variety of quality measures.

More about Solving Problems

By default when the method solve is called, IloCplex first presolves the model; that is, it transforms the model into a smaller, yet equivalent model. This operation can be controlled with the following parameters:
- IloCplex::PreInd,
- IloCplex::PreDual,
- IloCplex::AggInd, and
- IloCplex::AggFill.

For the rare occasion when a user wants to monitor progress during presolve, the callback class IloCplex::PresolveCallbackI is provided.

After the presolve is completed, IloCplex solves the first node relaxation and (in cases of a true MIP) enters the branch & cut process. IloCplex provides callback
classes that allow the user to monitor solution progress at each level. Callbacks derived from `IloCplex::ContinuousCallbackI` or one of its derived classes are called regularly during the solution of a node relaxation (including the root), and callbacks derived from `IloCplex::MIPCallbackI` or one of its derived callbacks are called regularly during branch & cut search. All callbacks provide the option to abort the current optimization.

**Branch Priorities and Directions**

When a branch occurs at a node in the branch & cut tree, usually there is a set of fractional-valued variables available to pick from for branching. `IloCplex` has several built-in rules for making such a choice, and they can be controlled by the parameter `IloCplex::VarSel`. Also, the method `setPriority` allows the user to specify a priority order. An instance of `IloCplex` branches on variables with an assigned priority before variables without a priority. It also branches on variables with higher priority before variables with lower priority, when the variables have fractional values.

Frequently, when two new nodes have been created (controlled by the parameter `IloCplex::BtTol`), one of the two nodes is processed next. This activity is known as diving. The branch direction determines which of the branches, the up or the down branch, is used when diving. By default, `IloCplex` automatically selects the branch direction. The user can control the branch direction by the method `setDirection`.

As mentioned before, the greatest flexibility for controlling the branching during branch & cut search is provided through goals (see `IloCplex::Goal`) or through the callbacks (see `IloCplex::BranchCallbackI`). With these concepts, you can control the branching decision based on runtime information during the search, instead of statically through branch priorities and directions, but the default strategies work well on many problems.

**Cuts**

An instance of `IloCplex` can also generate certain cuts in order to strengthen the relaxation, that is, in order to make the relaxation a better approximation of the original MIP. Cuts are constraints added to a model to restrict (cut away) noninteger solutions that would otherwise be solutions of the relaxation. The addition of cuts usually reduces the number of branches needed to solve a MIP.

When solving a MIP, `IloCplex` tries to generate violated cuts to add to the problem after solving a node. After `IloCplex` adds cuts, the subproblem is re-optimized. `IloCplex` then repeats the process of adding cuts at a node and reoptimizing until it finds no further effective cuts.

An instance of `IloCplex` generates its cuts in such a way that they are valid for all subproblems, even when they are discovered during analysis of a particular node. After a cut has been added to the problem, it will remain in the problem to the end of the optimization. However, cuts are added only internally; that is, they will not be part of the model extracted to the `IloCplex` object after the optimization. Cuts are most
frequently seen at the root node, but they may be added by an instance of IloCplex at other nodes as conditions warrant.

IloCplex looks for various kinds of cuts that can be controlled by the following parameters:

- IloCplex::Cliques,
- IloCplex::Covers,
- IloCplex::FlowCovers,
- IloCplex::GUBCovers,
- IloCplex::FracCuts,
- IloCplex::MIRCuts,
- IloCplex::FlowPaths,
- IloCplex::ImplBd, and
- IloCplex::DisjCuts.

During the search, you can query information about those cuts with a callback (see IloCplex::MIPCallbackI and its subclasses). For types of cuts that may take a long time to generate, callbacks are provided to monitor the progress and potentially abort the cut generation progress. In particular, those callback classes are IloCplex::FractionalCutCallbackI and IloCplex::DisjunctiveCutCallbackI. The callback class IloCplex::CutCallbackI allows you to add your own problem-specific cuts during search. This callback also allows you to generate and add local cuts, that is cuts that are only valid within the subtree where they have been added.

Instead of using callbacks, you can use goals to add your own cuts during the optimization.

**Heuristics**

After a node has been processed, that is, the LP has been solved and no more cuts were generated, IloCplex may try to construct an integer feasible solution from the LP solution at that node. The parameter IloCplex::HeurFreq and other parameters provide some control over this activity. In addition, goals or the callback class IloCplex::HeuristicCallbackI make it possible to call user-written heuristics to find an integer feasible solution.

Again, instead of using callbacks, you can use goals to add inject your own heuristically constructed solution into the running optimization.

**Node Selection**
When IloCplex is not diving but picking an unexplored node from the tree, several options are available that can be controlled with the parameter IloCplex::NodeSel. Again, IloCplex offers a callback class, IloCplex::NodeCallbackI, to give the user full control over this selection. With goals, objects of type IloCplex::NodeEvaluatorI can be used to define your own selection strategy.

See also IloAlgorithm in the ILOG Concert Reference Manual.

See also Goals among the Concepts in this manual. See also goals in the ILOG CPLEX User's Manual.

See Also

IloCplex::Algorithm, IloCplex::BasisStatus, IloCplex::BasisStatusArray, IloCplex::BranchDirection, IloCplex::BranchDirectionArray, IloCplex::CallbackI, IloCplex::DeleteMode, IloCplex::DualPricing, IloCplex::Exception, IloCplex::IISStatus, IloCplex::IISStatusArray, IloCplex::IntParam, IloCplex::MIPEmphasisType, IloCplex::NodeSelect, IloCplex::NumParam, IloCplex::PrimalPricing, IloCplex::Quality, IloCplex::CplexStatus, IloCplex::StringParam, IloCplex::VariableSelect, IloCplex::GoalI

Constructors

public IloCplex(IloEnv env)

This constructor creates an ILOG CPLEX algorithm. The new IloCplex object has no IloModel loaded (or extracted) to it.

public IloCplex(const IloModel model)

This constructor creates an ILOG CPLEX algorithm and extracts model for that algorithm.

When you create an algorithm (an instance of IloCplex, for example) and extract a model for it, you can write either this line:

IloCplex cplex(model);

or these two lines:

IloCplex cplex(env);
cplex.extract(model);

Methods

public IloConstraint addCut(IloConstraint con)
This method adds \texttt{con} as a cut to the invoking \texttt{IloCplex} object. The cut is not extracted as the regular constraints in a model, but is only copied when invoking the method \texttt{addCut}. Thus, \texttt{con} may be deleted or modified after \texttt{addCut} has been called and the change will not be notified to the invoking \texttt{IloCplex} object.

When columns are deleted from the extracted model, all cuts are deleted as well and need to be reextracted if they should be considered. Cuts are not part of the root problem, but are considered on an as-needed basis. A solution computed by \texttt{IloCplex} is guaranteed to satisfy all cuts added with this method.

\texttt{public const IloConstraintArray addCuts(const IloConstraintArray con)}

This method adds the constraints in \texttt{con} as cuts to the invoking \texttt{IloCplex} object. Everything said for \texttt{addCut} applies equally to each of the cuts given in array \texttt{con}.

\texttt{public IloConstraint addLazyConstraint(IloConstraint con)}

\textbf{Note:} This is an advanced method. Advanced methods typically demand a profound understanding of the algorithms used by ILOG CPLEX. Thus they incur a higher risk of incorrect behavior in your application, behavior that can be difficult to debug. Therefore, ILOG encourages you to consider carefully whether you can accomplish the same task by means of other methods instead.

This method adds \texttt{con} as a lazy constraint to the invoking \texttt{IloCplex} object. The constraint \texttt{con} is copied into the lazy constraint pool; the \texttt{con} itself is not part of the pool, so changes to \texttt{con} after it has been copied into the lazy constraint pool will not affect the lazy constraint pool.

Lazy constraints added with \texttt{addLazyConstraint} are typically constraints of the model that are not expected to be violated when left out. The idea behind this is that the LPs that are solved when solving the MIP can be kept smaller when these constraints are not included. \texttt{IloCplex} will, however, include a lazy constraint in the LP as soon as it becomes violated. In other words, the solution computed by \texttt{IloCplex} makes sure that all the lazy constraints that have been added are satisfied.

By contrast, if the constraint does not change the feasible region of the extracted model but only strengthens the formulation, it is referred to as a user cut. While user cuts can be added to \texttt{IloCplex} with \texttt{addLazyConstraint}, it is generally preferable to do so with \texttt{addUserCuts}. It is an error, however, to add lazy constraints by means of the method \texttt{addUserCuts}.

When columns are deleted from the extracted model, all lazy constraints are deleted as well and need to be recopied into the lazy constraint pool. Use of this method in place of \texttt{addCuts} allows for further presolve reductions.

This method is equivalent to \texttt{addCut}.
public const IloConstraintArray addLazyConstraints(const IloConstraintArray con)

**Note:** This is an advanced method. Advanced methods typically demand a profound understanding of the algorithms used by ILOG CPLEX. Thus they incur a higher risk of incorrect behavior in your application, behavior that can be difficult to debug. Therefore, ILOG encourages you to consider carefully whether you can accomplish the same task by means of other methods instead.

This method adds a set of lazy constraints to the invoking IloCplex object. Everything said for addLazyConstraint applies to each of the lazy constraints given in array con.

This method is equivalent to addCuts.

public IloConstraint addUserCut(IloConstraint con)

**Note:** This is an advanced method. Advanced methods typically demand a profound understanding of the algorithms used by ILOG CPLEX. Thus they incur a higher risk of incorrect behavior in your application, behavior that can be difficult to debug. Therefore, ILOG encourages you to consider carefully whether you can accomplish the same task by means of other methods instead.

This method adds con as a user cut to the invoking IloCplex object. The constraint con is copied into the user cut pool; the con itself is not part of the pool, so changes to con after it has been copied into the user cut pool will not affect the user cut pool.

Cuts added with addUserCut must be real cuts, in that the solution of a MIP does not depend on whether the cuts are added or not. Instead, they are there only to strengthen the formulation.

When columns are deleted from the extracted model, all user cuts are deleted as well and need to be recopied into the user cut pool.

**Note:** It is an error to use addUserCut for lazy constraints, that is, constraints whose absence may potentially change the solution of the problem. Use addLazyConstraints or, equivalently, addCut when you add such a constraint.

public const IloConstraintArray addUserCuts(const IloConstraintArray con)
This method adds a set of user cuts to the invoking `IloCplex` object. Everything said for `addUserCut` applies to each of the user cuts given in array `con`.

```java
public static IloCplex::Goal Apply(IloCplex cplex,
IloCplex::Goal goal,
IloCplex::NodeEvaluator eval)
```

This method is used to create and return a goal that applies the node selection strategy defined by `eval` to the search strategy defined by `goal`. The resulting goal will use the node strategy defined by `eval` for the subtree generated by `goal`.

```java
public void basicPresolve(const IloNumVarArray vars,
IloNumArray redlb,
IloNumArray redub,
const IloRangeArray rngs,
IloBoolArray redundant)
```

This method can be used to compute tighter bounds for the model variables and to detect redundant constraints in the model extracted to the invoking `IloCplex` object. For every variable specified in parameter `vars`, it will return possibly tightened bounds in the corresponding elements of arrays `redlb` and `redub`. Similarly, for every constraint specified in parameter `rngs`, this method will return a boolean indicating whether or not it is redundant in the model in the corresponding element of array `redundant`.

```java
public void clearCuts()
```

This method deletes all cuts that have previously been added to the invoking `IloCplex` object with the methods `addCut` and `addCuts`.

```java
public void clearLazyConstraints()
```

**Note:** This is an advanced method. Advanced methods typically demand a profound understanding of the algorithms used by ILOG CPLEX. Thus they incur a higher risk of incorrect behavior in your application, behavior that can be difficult to debug. Therefore, ILOG encourages you to consider carefully whether you can accomplish the same task by means of other methods instead.

**Note:** This is an advanced method. Advanced methods typically demand a profound understanding of the algorithms used by ILOG CPLEX. Thus they incur a higher risk of incorrect behavior in your application, behavior that can be difficult to debug. Therefore, ILOG encourages you to consider carefully whether you can accomplish the same task by means of other methods instead.
This method deletes all lazy constraints added to the invoking \texttt{IloCplex} object with the methods \texttt{addLazyConstraint} and \texttt{addLazyConstraints}.

This method is equivalent to \texttt{clearCuts}.

\begin{verbatim}
public void clearModel()
\end{verbatim}

This method can be used to unextract the model that is currently extracted to the invoking \texttt{IloCplex} object.

\begin{verbatim}
public void clearUserCuts()
\end{verbatim}

\textbf{Note:} This is an advanced method. Advanced methods typically demand a profound understanding of the algorithms used by ILOG CPLEX. Thus they incur a higher risk of incorrect behavior in your application, behavior that can be difficult to debug. Therefore, ILOG encourages you to consider carefully whether you can accomplish the same task by means of other methods instead.

This method deletes all user cuts that have previously been added to the invoking \texttt{IloCplex} object with the methods \texttt{addUserCut} and \texttt{addUserCuts}.

\begin{verbatim}
public void delDirection(IloNumVar var)
\end{verbatim}

This method removes any existing branching direction assignment from variable \texttt{var}.

\begin{verbatim}
public void delDirections(const IloNumVarArray var)
\end{verbatim}

This method removes any existing branching direction assignments from all variables in the array \texttt{var}.

\begin{verbatim}
public void delPriorities(const IloNumVarArray var)
\end{verbatim}

This method removes any existing priority order assignments from all variables in the array \texttt{var}.

\begin{verbatim}
public void delPriority(IloNumVar var)
\end{verbatim}

This method removes any existing priority order assignment from variable \texttt{var}.

\begin{verbatim}
public IloBool feasOpt(const IloRangeArray rngs,
                      IloNumArray rnglb,
                      IloNumArray rngub,
                      IloBool optimize)
\end{verbatim}

\begin{verbatim}
Note: This is an advanced method. Advanced methods typically demand a profound understanding of the algorithms used by ILOG CPLEX. Thus they incur a higher risk of incorrect behavior in your application, behavior that can be difficult to debug. Therefore, ILOG encourages you to consider carefully whether you can accomplish the same task by means of other methods instead.
\end{verbatim}

This method writes the model that has been extracted to the invoking algorithm in the file \texttt{filename}. The name of the file must include an extension that ILOG CPLEX recognizes as a file format for a model. Supported formats are .sav, .lp and .mps. See the reference manual \textit{ILOG CPLEX File Formats} for more detail and the \textit{ILOG CPLEX User's Manual} for additional information about file formats.

\begin{verbatim}
public IloBool feasOpt(const IloRangeArray rngs,
                      IloNumArray rnglb,
                      IloNumArray rngub,
                      IloBool optimize)
\end{verbatim}
Attempts to find a feasible relaxation of an active model by relaxing the bounds of the constraints specified in `rngs` as specified in `rnglb` and `rngub`. This is a special version of the method `feasOpt(IloRangeArray, IloNumArray, IloNumVarArray, IloNumArray, IloNumArray, IloBool)`

The method returns `IloTrue` if a feasible relaxation is found along with a solution for it.

```java
public IloBool feasOpt(const IloNumVarArray vars,
                       IloNumArray varlb,
                       IloNumArray varub,
                       IloBool optimize)
```

Attempts to find a feasible relaxation of an active model by relaxing the bounds of the variables specified in `vars` as specified in `varlb` and `varub`. This is a special version of the method `feasOpt(IloRangeArray, IloNumArray, IloNumArray, IloNumVarArray, IloNumArray, IloNumArray, IloBool)`

The method returns `IloTrue` if a feasible relaxation is found, along with a solution for it.

```java
public IloBool feasOpt(const IloRangeArray rngs,
                       IloNumArray rnglb,
                       IloNumArray rngub,
                       const IloNumVarArray vars,
                       IloNumArray varlb,
                       IloNumArray varub,
                       IloBool optimize)
```

The method `feasOpt` computes a minimal relaxation of the bounds of variables and constraints that make the active model feasible. For each bound, the user may specify a preferred value indicating how much a relaxation of that bound is acceptable. A negative or 0 (zero) value indicates that the corresponding bound must not be relaxed. Typically, values greater than or equal to 1 (one) should be used. The preference value for relaxing the lower and upper bound of constraint `rngs[i]` must be provided in `rnglb[i]` and `rngub[i]`, respectively. Similarly, the preference value for relaxing the lower and upper bound of constraint `vars[i]` must be provided in `varlb[i]` and `varub[i]`, respectively.

If enough variables or constraints were allowed to be relaxed, the function will return `IloTrue` and `IloFalse` otherwise. In case of success, the routine returns with suggested relaxed values that would make the active model feasible. These bounds are chosen in such a way that the relaxation is minimal in the sense that the sum of (relaxation amount)/(preference value) over all variables and constraints is minimized.

The bounds that make the model feasible are returned in the same arrays as the preference values are passed into this method. That is, after successful termination of the method `feasOpt`, `rnglb[i]` and `rngub[i]` contain the lower and upper bound.
to use for the constraint \( \text{rngs}[i] \) in order to make the model feasible. Similarly, \( \text{varlb}[i] \) and \( \text{varub}[i] \) contain the lower and upper bound to use for variable \( \text{vars}[i] \) in order to make the model feasible. If bounds were not allowed to be relaxed or did not need to be relaxed in order to make the model feasible, the original bounds are returned.

The active model is not changed by this method.

If \( \text{feasOpt} \) successfully finds a relaxation that made the model feasible, it also installs a solution vector that would be feasible, had the model been relaxed as specified with the returned relaxed bounds. This solution can be queried with the conventional query methods.

When the \( \text{optind} \) parameter is \( \text{IloTrue} \), this method also tries to find the optimal solution of the minimally relaxed model. That is, it finds the solution that is optimal with respect to the original objective function, but requires only minimal relaxation. With the default setting of the parameter \( \text{optind} \) the original objective function is not evaluated.

```java
public void freePresolve()
```

This method frees the presolved problem. Under the default setting of parameter \( \text{Reduce} \), the presolved problem is freed when an optimal solution is found; however, it is not freed if \( \text{Reduce} \) has been set to 1 (primal reductions) or to 2 (dual reductions). In these instances, the function \( \text{freePresolve} \) can be used when necessary to free it manually.

```java
public IloCplex::Algorithm getAlgorithm()
```

This method returns the algorithm type that was used to solve the most recent model in cases where it was not a MIP.

```java
public void getAX(IloNumArray val,
                   const IloRangeArray con)
```

For the constraints in \( \text{con} \), this method places the values of the expressions, or, equivalently, the activity levels of the constraints for the current solution of the invoking \( \text{IloCplex} \) object into the array \( \text{val} \). Array \( \text{val} \) is resized to the same size as array \( \text{con} \), and \( \text{val}[i] \) will contain the slack value for constraint \( \text{con}[i] \). All ranges in \( \text{con} \) must be part of the extracted model.

```java
public IloNum getAX(const IloRange range)
```

This method returns the value of the expression of constraint \( \text{range} \), or, equivalently, its activity level, for the current solution of the invoking \( \text{IloCplex} \) object. The range must be part of the extracted model.

```java
public IloCplex::BasisStatus getBasisStatus(const IloConstraint con)
```

This method returns the basis status of the implicit slack or artificial variable created for the constraint \( \text{con} \).
public IloCplex::BasisStatus getBasisStatus(const IloIntVar var)

This method returns the basis status for the variable var.

public IloCplex::BasisStatus getBasisStatus(const IloNumVar var)

This method returns the basis status for the variable var.

public void getBasisStatuses(IloCplex::BasisStatusArray cstat,
const IloNumVarArray var,
IloCplex::BasisStatusArray rstat,
const IloConstraintArray con)

This method puts the basis status of each variable in var into the corresponding element of the array cstat, and it puts the status of each row in con (an array of ranges or constraints) into the corresponding element of the array rstat. Arrays rstat and cstat are resized accordingly.

public void getBasisStatuses(IloCplex::BasisStatusArray stat,
const IloConstraintArray con)

This method puts the basis status of each constraint in con into the corresponding element of the array stat. Array stat is resized accordingly.

public void getBasisStatuses(IloCplex::BasisStatusArray stat,
const IloNumVarArray var)

This method puts the basis status of each variable in var into the corresponding element of the array stat. Array stat is resized accordingly.

public IloNum getBestObjValue()}

This method returns a bound on the optimal solution value of the problem. When a model has been solved to optimality, this value matches the optimal solution value. If a MIP optimization is terminated before optimality has been proven, this value is computed for a minimization (maximization) problem as the minimum (maximum) objective function value of all remaining unexplored nodes.

public void getBoundSA(IloNumArray lblower,
IloNumArray lbupper,
IloNumArray ublower,
IloNumArray ubupper,
const IloNumVarArray vars)

For the given set of variables vars, bound sensitivity information is computed. When the method returns, the element lblower[j] and lbupper[j] will contain the lowest and highest value the lower bound of variable vars[j] can assume without affecting the optimality of the solution. Likewise, ublower[j] and ubupper[j] will contain the lowest and highest value the upper bound of variable vars[j] can assume without affecting the optimality of the solution. The arrays lblower, lbupper, ublower, and ubupper will be resized to the size of array vars. The value 0 (zero) can be passed for any of the return arrays if the information is not desired.
public IloCplex::CplexStatus getCplexStatus()

This method returns the ILOG CPLEX status of the invoking algorithm. For possible ILOG CPLEX values, see the enumeration type IloCplex::CplexStatus.

public IloCplex::CplexStatus getCplexSubStatus()

This method is used to access the solution status of the last node problem that was solved in the event of an error termination in the previous invocation of solve. The method IloCplex::getCplexSubStatus returns 0 in the event of a normal termination. If the invoking IloCplex object is continuous, this is equivalent to the status returned by the method getCplexStatus.

public IloNum getCutoff()

This method returns the MIP cutoff value being used during the MIP optimization. In a minimization problem, all nodes are pruned that have an optimal solution value of the continuous relaxation that is larger than the current cutoff value. The cutoff is updated with the incumbent. If the invoking IloCplex object is an LP or QP, +IloInfinity or -IloInfinity is returned, depending on the optimization sense.

public IloBool getDefault(IloCplex::BoolParam parameter)

These method return the default setting for the parameter parameter.

public IloCplex::DeleteMode getDeleteMode()

This method returns the current delete mode of the invoking IloCplex object.

public IloCplex::BranchDirection getDirection(IloNumVar var)

This method returns the branch direction previously assigned to variable var with method setDirection or setDirections. If no direction has been assigned, IloCplex::BranchGlobal will be returned.

public void getDirections(IloCplex::BranchDirectionArray dir, const IloNumVarArray var)

This method returns the branch directions previously assigned to variables listed in var with the method setDirection or setDirections. When the function returns, dir[i] will contain the branch direction assigned for variables var[i]. If no branch direction has been assigned to var[i], dir[i] will be set to IloCplex::BranchGlobal.

public IloExtractable getDiverging()

This method returns the diverging variable or constraint, in a case where the primal Simplex algorithm has determined the problem to be infeasible. The returned extractable is either an IloNumVar or an IloConstraint object extracted to the invoking IloCplex optimizer; it is of type IloNumVar if the diverging column corresponds to a variable, or of type IloConstraint if the diverging column corresponds to the slack variable of a constraint.
public IloNum getDual(const IloRange range)

This method returns the dual value associated with constraint range in the current solution of the invoking algorithm.

public void getDuals(IloNumArray val, const IloRangeArray con)

This method puts the dual values associated with the ranges in the array con into the array val. Array val is resized to the same size as array con, and val[i] will contain the dual value for constraint con[i].

public IloNum getFormulationBigValue()

The method getFormulationBigValue returns the current setting of the large value used to set a bound on otherwise unbounded expressions appearing in logical constraints. Such a value is sometimes referred to as Big M.

By default, this value is 1e6. You can change that default setting by means of the method setFormulationBigValue.

In order to transform a logical constraint automatically into an equivalent MIP representation, it is sometimes necessary to set a finite upper or lower bound on an otherwise unbounded term. When such a bound does not already exist, CPLEX uses the value returned by this method for that purpose. For example, consider the following disjunction (that is, or-statement):

\[(x == 1) \text{ || } (x \neq 2)\]

It is necessary to have an upper bound on x. If the upper bound on x is IloInfinity, CPLEX will use the value returned by getFormulationBigValue in the transformed representation of logical formulations.

public IloNum getFormulationEpsValue()

This method returns the epsilon value that is used to model strict relations. In order to model a strict inequality, such as \(x < b\) in a MIP, IloCplex uses \(x \leq b - \text{eps}\) as a surrogate, where eps is the value returned by this method.

public void getIIS(IloCplex::IISStatusArray cstat, IloNumVarArray var, IloCplex::IISStatusArray rstat, IloConstraintArray rng)

If an LP model has been proven to be infeasible, this method can compute an irreducibly inconsistent set (IIS). The variables and range constraints in the IIS are returned in the arrays var and rng, respectively.
For each variable \( \text{var}[i] \), \( \text{cstat}[i] \) gives the IIS status for that variable in the IIS, where the meaning of \( \text{cstat}[i] \) is:
- \( \text{IloCplex::Lower} \) the lower bound of the variable participates in the IIS
- \( \text{IloCplex::Upper} \) the upper bound of the variable participates in the IIS
- \( \text{IloCplex::Both} \) both variable bounds participate in the IIS

Similarly, for each range constraint \( \text{rng}[i] \), \( \text{rstat}[i] \) gives the IIS status for that constraint in the IIS, where the meaning of \( \text{rstat}[i] \) is:
- \( \text{IloCplex::Lower} \) the lower bound of the constraint participates in the IIS
- \( \text{IloCplex::Upper} \) the upper bound of the constraint participates in the IIS
- \( \text{IloCplex::Both} \) range is equality constraint and participates in the IIS.

The statuses \( \text{IloCplex::Lower} \) and \( \text{IloCplex::Upper} \) are needed only for constraints that are truly ranged, that is, where both bounds are finite but different.

By definition, if any of the bounds or constraints participating in the IIS is removed, the IIS becomes feasible. However, this does not mean that the model also becomes feasible by doing so. Instead, there may be yet another source of infeasibility in the model.

If the LP has been proven infeasible but no basis is available, \text{IloCplex} automatically invokes the primal simplex optimizer to compute the basis needed for the IIS-finding algorithm.

```java
public IloInt getIncumbentNode()
```
This method returns the node number where the current incumbent was found. If the invoking \text{IloCplex} object is an LP or a QP, 0 is returned.

```java
public IloInt getMax(IloCplex::IntParam parameter)
```
These method return the maximum allowed value for the parameter \( \text{parameter} \).

```java
public IloInt getMin(IloCplex::IntParam parameter)
```
These method return the minimum allowed value for the parameter \( \text{parameter} \).

```java
public IloInt getNbarrierIterations()
```
This method returns the number of barrier iterations from the last solve.

```java
public IloInt getNbinVars()
```
This method returns the number of binary variables in the matrix representation of the active model in the invoking \text{IloCplex} object.
public IloInt getNcliques()

This method returns the number of clique cuts that have been added to the problem. If the invoking IloCplex object is not a MIP, 0 is returned.

public IloInt getNcols()

This method returns the number of columns extracted for the invoking algorithm. There may be differences in the number returned by this function and the number of object of type IloNumVar and its subclasses in the model that is extracted. This is because automatic transformation of non-linear constraints and expressions may introduce new variables.

public IloInt getNcovers()

This method returns the number of cover cuts that have been added to the problem. If the invoking IloCplex object is not a MIP, 0 is returned.

public IloInt getNcrossDExch()

This method returns the number of dual exchange operations in the crossover of the last call to method solve or solveFixed, if barrier with crossover has been used for solving an LP or QP.

public IloInt getNcrossDPush()

This method returns the number of dual push operations in the crossover of the last call to solve or solveFixed, if barrier with crossover was used for solving an LP or QP.

public IloInt getNcrossPExch()

This method returns the number of primal exchange operations in the crossover of the last call of method solve or solveFixed, if barrier with crossover was used for solving an LP of QP.

public IloInt getNcrossPPush()

This method returns the number of primal push operations in the crossover of the last call of method solve or solveFixed, if barrier with crossover was used for solving an LP or QP.

public IloInt getNdualSuperbasics()

This method returns the number of dual superbasic variables in the current solution of the invoking IloCplex object.

public IloInt getNintVars()

This method returns the number of integer variables in the matrix representation of the active model in the invoking IloCplex object.

public IloInt getNiterations()
This method returns the number of iterations that occurred during the last call to the method `solve` in the invoking algorithm.

```java
public IloInt getNnodes()
```

This method returns the number of branch-and-cut nodes that were processed in the current solution. If the invoking `IloCplex` object is not a MIP, it returns 0.

```java
public IloInt getNnodesLeft()
```

This method returns the number of branch-and-cut nodes that remain to be processed in the current solution. If the invoking `IloCplex` object is not a MIP, it returns 0.

```java
public IloInt getNNZs()
```

This method returns the number of non-zeros extracted to the constraint matrix A of the invoking algorithm.

```java
public IloInt getNphaseOneIterations()
```

If a simplex method was used for solving continuous model, this method returns the number of iterations in phase 1 of the last call to `solve` or `solveFixed`.

```java
public IloInt getNprimalSuperbasics()
```

This method returns the number of primal superbasic variables in the current solution of the invoking `IloCplex` object.

```java
public IloInt getNQCs()
```

This method returns the number of quadratic constraints extracted from the active model for the invoking algorithm. This number may be different from the total number of constraints in the active model because linear constraints are not accounted for in this function.

**See Also**

- `getNrows`

```java
public IloInt getNrows()
```

This method returns the number of rows extracted for the invoking algorithm. There may be differences in the number returned by this function and the number of `IloRanges` and `IloConstraints` in the model that is extracted. This is because quadratic constraints are not accounted for by method `getNrows` and automatic transformation of non-linear constraints and expressions may introduce new constraints.

**See Also**

- `getNQCs`

```java
public IloInt getNsemiContVars()
```

This method returns the number of semi-continuous variables in the matrix representation of the active model in the invoking `IloCplex` object.

```java
public IloInt getNsemiIntVars()
```
This method returns the number of semi-integer variables in the matrix representation of
the active model in the invoking IloCplex object.

```java
public IloInt getNsiftingIterations()
```

This method returns the number of sifting iterations performed for solving the last LP
with algorithm type IloCplex::Sifting, or, equivalently, the number of work LPs
that have been solved for it.

```java
public IloInt getNsiftingPhaseOneIterations()
```

This method returns the number of sifting iterations performed for solving the last LP
with algorithm type IloCplex::Sifting in order to achieve primal feasibility.

```java
public IloInt getNSOSs()
```

This method returns the number of SOSs extracted for the invoking algorithm. There
may be differences in the number returned by this function and the number of
IloNumVar, etc. in the model that is extracted. This is because piecewise linear
functions are extracted to a set of SOS.

```java
public void getObjSA(IloNumArray lower,
                     IloNumArray upper,
                     const IloNumVarArray vars)
```

This method performs objective sensitivity analysis for the variables specified in array
vars. When this method returns lower[i] and upper[i] will contain the lowest
and highest value the objective function coefficient for variable vars[i] can assume
without affecting the optimality of the solution. The arrays lower and upper will be
resized to the size of array vars. If any of the information is not requested, 0 (zero) can
be passed for the corresponding array parameter.

```java
public IloBool getParam(IloCplex::BoolParam parameter)
```

This method returns the current setting of parameter in the invoking algorithm.

See the reference manual ILOG CPLEX Parameters and the ILOG CPLEX User's Manual for more information about these parameters. Also see the user's manual for
examples of their use.

```java
public void getPriorities(IloNumArray pri,
                          const IloNumVarArray var)
```

This method returns query branch priorities previously assigned to variables listed in
var with the method setPriority or setPriorities. When the function returns, pri[i] will contain the priority value assigned for variables var[i]. If no priority has been assigned to var[i], pri[i] will contain 0 (zero).

```java
public IloNum getPriority(IloNumVar var)
```

This method returns the priority previously assigned to the variable var with the
method setPriority or setPriorities. It returns 0 (zero) if no priority has been assigned.
public IloCplex::PWLFormulation getPWLFormulation()

This method queries the way piecewise linear expressions are represented when CPLEX automatically transforms them during extraction.

public IloNum getQuality(IloCplex::Quality q,
                   IloNumVar * var,
                   IloConstraint * rng)

These method return the requested quality measure.

Some quality measures are related to a variable or a constraint. For example IloCplex::MaxPrimalInfeas is related to the variable or range constraint where the maximum infeasibility (bound violation) occurs. If this information is also requested, pointers to IloNumVar or IloConstraint objects may be passed for the optional parameters where the relevant variable or range will be written.

public void getRangeSA(IloNumArray lblower,
                    IloNumArray lbupper,
                    IloNumArray ublower,
                    IloNumArray ubupper,
                    const IloRangeArray con)

This method performs sensitivity analysis for the upper and lower bounds of the ranged constraints passed in the array con. When the method returns, lblower[i] and lbupper[i] will contain, respectively, the lowest and the highest value that the lower bound of constraint con[i] can assume without affecting the optimality of the solution. Similarly, ublower[i] and ubupper[i] will contain, respectively, the lowest and the highest value that the upper bound of the constraint con[i] can assume without affecting the optimality of the solution. The arrays lblower, lbupper, ublower, and ubupper will be resized to the size of array con. If any of the information is not requested, 0 can be passed for the corresponding array parameter.

public void getRay(IloNumArray vals,
                  IloNumVarArray vars)

This method returns an unbounded direction (also known as a ray) corresponding to the present basis for an LP that has been determined to be an unbounded problem. CPLEX puts the variables of the extracted model into the array vars and it puts the corresponding values of the unbounded direction into the array vals.

Note: CPLEX resizes these arrays for you.

public IloNum getReducedCost(const IloIntVar var)

This method returns the reduced cost associated with var in the invoking algorithm.

public IloNum getReducedCost(const IloNumVar var)
This method returns the reduced cost associated with var in the invoking algorithm.

```java
public void getReducedCosts(IloNumArray val,
                             const IloIntVarArray var)
```

This method puts the reduced costs associated with the numeric variables of the array var into the array val. The array val is automatically resized to the same length as array var, and val[i] will contain the reduced cost for variable var[i].

```java
public void getReducedCosts(IloNumArray val,
                             const IloNumVarArray var)
```

This method puts the reduced costs associated with the variables in the array var into the array val. Array val is resized to the same size as array var, and val[i] will contain the reduced cost for variable var[i].

```java
public void getRHSSA(IloNumArray lower,
                      IloNumArray upper,
                      const IloRangeArray cons)
```

This method performs righthand side sensitivity analysis for the constraints specified in array cons. The constraints must be of the form cons[i]: expr[i] rel rhs[i]. When this method returns lower[i] and upper[i] will contain the lowest and highest value rhs[i] can assume without affecting the optimality of the solution. The arrays lower and upper will be resized to the size of array cons. If any of the information is not requested, 0 (zero) can be passed for the corresponding array parameter.

```java
public IloNum getSlack(const IloRange range)
```

This method returns the slack value associated with constraint range in the current solution of the invoking algorithm. For a range with finite lower and upper bounds, the slack value consists of the difference between the expression of the range and its lower bound.

```java
public void getSlacks(IloNumArray val,
                       const IloRangeArray con)
```

This method puts the slack values associated with the constraints indicated by the array con into the array val. For a ranged constraint with finite lower and upper bounds, the slack value consists of the difference between the expression in the range and its lower bound. Array val is resized to the same size as array con, and val[i] will contain the slack value for constraint con[i].

```java
public IloAlgorithm::Status getStatus()()
```

This method returns the status of the invoking algorithm. For its ILOG CPLEX status, see the method IloCplex::getCplexStatus.

```java
public IloCplex::Algorithm getSubAlgorithm()
```

This method returns the algorithm type that was used to solve most recent node of a MIP.
public void **getValues** (const IloIntVarArray var,  
IloNumArray val)

This method puts the solution values of the integer variables indicated by the array var into the array val. Array val is resized to the same size as array var, and val[i] will contain the solution value for variable var[i].

public void **getValues** (IloNumArray val,  
const IloIntVarArray var)

This method puts the solution values of the integer variables indicated by the array var into the array val. Array val is resized to the same size as array var, and val[i] will contain the solution value for variable var[i].

public void **getValues** (IloNumArray val,  
const IloNumVarArray var)

This method puts the solution values of the numeric variables indicated by the array var into the array val. Array val is resized to the same size as array var, and val[i] will contain the solution value for variable var[i].

public const char * **getVersion**()

This method returns a string indicating the version of IloCplex.

public void **importModel** (IloModel & m,  
const char * filename)

This method reads a model from the file indicated by filename into model. The invoking IloCplex object is not affected when calling this method unless model is its active model.

public void **importModel** (IloModel & m,  
const char * filename,  
IloObjective & obj,  
IloNumVarArray vars,  
IloRangeArray rngs,  
IloRangeArray lazy,  
IloRangeArray cuts)

This method is an simplification of the importModel method that does not provide arrays to return SOS’s. This method is more easy to use in the case that you are dealing with continuous models, because in this case you already know that no SOS’s will be present.

public void **importModel** (IloModel & model,  
const char * filename,  
IloObjective & obj,  
IloNumVarArray vars,  
IloRangeArray rngs,  
IloSOS1Array sos1,  
IloSOS2Array sos2,  
IloRangeArray lazy,  
IloRangeArray cuts)
This method reads a model from the file indicated by filename into model. Note, that the invoking IloCplex object is not affected when calling this method unless model is its active model. It places the objective it has read in obj, the variables it has read in the array vars, the ranges it has read in the array rngs; and the Special Ordered Sets it has read in the arrays sos1 and sos2.

Note: CPLEX resizes these arrays for you to accommodate the returned objects.

Note: This note is for advanced users only. The two arrays lazy and cuts are filled with the lazy constraints and user cuts that may be included in the model in file filename.

public IloBool isDualFeasible()

This method returns IloTrue if a dual feasible solution is recorded in the invoking IloCplex object and can be queried.

public IloBool isMIP()

This method returns IloTrue if the invoking algorithm has extracted a model that is a MIP (mixed-integer programming problem) and IloFalse otherwise. Member functions for accessing duals and reduced cost basis work only if the model is not a MIP.

public IloBool isPrimalFeasible()

This method returns IloTrue if a primal feasible solution is recorded in the invoking IloCplex object and can be queried.

public IloBool isQC()

This method returns IloTrue if the invoking algorithm has extracted a model that is quadratically constrained. Otherwise, it returns IloFalse. For an explanation of quadratically constrained see the topic QCP in the ILOG CPLEX User’s Manual.

public IloBool isQO()

This method returns IloTrue if the invoking algorithm has extracted a model that has quadratic objective function terms. Otherwise, it returns IloFalse.

public static IloCplex::Goal LimitSearch(IloCplex cplex, IloCplex::Goal goal, IloCplex::SearchLimit limit)
This method creates and returns a goal that puts the search specified by `goal` under the limit defined by `limit`. Only the subtree controlled by `goal` will be subjected to limit `limit`.

```java
public void presolve(IloCplex::Algorithm alg)
```

This method performs Presolve on the model. The enumeration `alg` tells Presolve which algorithm is intended to be used on the reduced model; `NoAlg` should be specified for MIP models.

```java
public void qpIndefCertificate(IloNumVarArray var,
      IloNumArray x)
```

The quadratic objective terms in a QP model must form a positive semi-definite Q matrix (negative semi-definite for maximization). If IloCplex finds that this is not true, it will discontinue the optimization. In such cases, the `qpIndefCertificate` method can be used to compute assignments (returned in array `x`) to all variables (returned in array `var`) such that the quadratic term of the objective function evaluates to a negative value \((x'Qx < 0\) in matrix terms) to prove the indefiniteness.

**Note:** CPLEX resizes these arrays for you.

```java
public void readBasis(const char * name)
```

Reads a simplex basis from the BAS file indicated by `name`, and copies that basis into the invoking IloCplex object. The parameter `AdvInd` must be set to a nonzero value (e.g. its default setting) for the simplex basis to be used to start a subsequent optimization with one of the Simplex algorithms.

By convention, the file extension is `.bas`. The BAS file format is documented in the reference manual *ILOG CPLEX File Formats*.

```java
public void readMIPStart(const char * name)
```

Reads the MST file indicated by `name` and copies the MIP start information into the invoking IloCplex object. The parameter `MIPStart` must be turned on in order for the MIP start information to be used to with a subsequent MIP optimization. The default value for the parameter is off.

By convention, the file extension is `.mst`. The MST file format is documented in the reference manual *ILOG CPLEX File Formats*.

```java
public void readOrder(const char * filename)
```

This method reads a priority order from an file in ORD format into the invoking IloCplex object. The names in the ORD file must match the names in the active model. The priority order will be associated with the model. The parameter
MipOrdType must be nonzero, for the next invocation of the method IloCplex::solve to take the order into account.

By convention, the file extension is .ord. The ORD file format is documented in the reference manual ILOG CPLEX File Formats.

public void readParam(const char * name)

Reads parameters and their settings from the file indicated by name and applies them to the invoking IloCplex object. Parameters not listed in the parameter file will be reset to their default setting.

By convention, the file extension is .prm. The PRM file format is documented in the reference manual ILOG CPLEX File Formats.

public void readTree(const char * name)

Reads branch-and-bound progress information from a prior run, contained in the TRE file indicated by name, into a CPLEX problem object. The parameter AdvInd must set to a nonzero value be on (e.g. its default value), in order for the tree to be used for starting a subsequent optimization.

By convention, the file extension is .tre. The TRE file format is documented in the reference manual ILOG CPLEX File Formats.

public void readVectors(const char * name)

Reads a barrier solution from the VEC file indicated by name and copies this information into a CPLEX problem object. This routine is typically used to initiate a crossover from the barrier solution. The parameter AdvInd must set to a nonzero value (e.g. its default setting) in order for the vector file to take effect for starting a crossover at the next invocation of a simplex algorithm with method solve.

By convention, the file extension is .vec. The VEC file format is documented in the reference manual ILOG CPLEX File Formats.

public void setBasisStatuses(const IloCplex::BasisStatusArray & cstat, const IloNumVarArray & var, const IloCplex::BasisStatusArray & rstat, const IloConstraintArray & con)

This method uses the array cstats to set the basis status of the variables in the array var; it uses the array rstats to set the basis status of the ranges in the array con.

public void setDefaults()

This method resets all CPLEX parameters to their default values.

public void setDeleteMode(IloCplex::DeleteMode mode)

This method sets the delete mode in the invoking IloCplex object to mode.

public void setDirection(IloNumVar var, IloCplex::BranchDirection dir)
This method sets the preferred branching direction for variable var to dir. This setting will cause CPLEX first to explore the branch indicated by dir after branching on variable var.

public void setDirections(const IloNumVarArray var,
                          const IloCplex::BranchDirectionArray dir)

This method sets the preferred branching direction for each variable in the array var to the corresponding value in the array dir. This will cause CPLEX to first explore the branch indicated by dir[i] after branching on variable var[i].

public void setFormulationBigValue(IloNum v)

This method sets the large value used to set a bound on any otherwise unbounded expressions appearing in logical constraints. Such a value is sometimes referred to as Big M. By default, it is set to 1e6.

The method getFormulationBigValue returns the current setting of the large value.

In order to transform a logical constraint automatically into an equivalent MIP formulation, it is sometimes necessary to set a finite upper or lower bound on an otherwise unbounded term. When such a bound does not already exist, CPLEX uses the value set by this method for that purpose. For example, consider the following disjunction (that is, an or-statement):

\[(x == 1) \lor (x \geq 2)\]

It is necessary to have an upper bound on x. If the upper bound on x is IloInfinity, CPLEX will use the value controlled by this method in the transformed representation of logical formulations.

public void setFormulationEpsValue(IloNum eps)

This method sets the epsilon value that is used to model strict relations. In order to model a strict inequality such as \(x < b\) in a MIP, IloCplex uses \(x \leq b - \text{eps}\) as a surrogate, where \(\text{eps}\) is the value controlled by this method.

public void setParam(IloCplex::BoolParam parameter,
                     IloBool value)

This method sets parameter to value in the invoking algorithm. See the ILOG CPLEX User's Manual for more detailed information about parameters and for examples of their use.

public void setPriorities(const IloNumVarArray var,
                          const IloNumArray pri)

This method sets the priority order for all variables in the array var to the corresponding value in the array pri. During branching, integer variables with higher
priorities are given preference over integer variables with lower priorities. Further, variables that have priority values assigned to them are given preference over variables that don't. Priorities must be positive integers.

```java
public void setPriority(IloNumVar var, IloNum pri)
```

This method sets the priority order for the variable `var` to `pri`. During branching, integer variables with higher priorities are given preference over integer variables with lower priorities. Further, variables that have priority values assigned to them are given preference over variables that don't. Priorities must be positive integers.

```java
public void setPWLFormulation(IloCplex::PWLFormulation f)
```

This method allows you to control the way piecewise linear expressions are represented when CPLEX automatically transforms them during extraction.

```java
public void setVectors(const IloNumArray x, const IloNumArray dj, const IloNumVarArray var, const IloNumArray slack, const IloNumArray pi, const IloRangeArray rng)
```

This method allows a user to specify a starting point for the following invocation of the `solve` method. Zero can be passed for any of the parameters. However, if `x` or `dj` is not zero, `var` must not be zero either. Similarly, if `slack` or `pi` is not zero, `rng` must not be zero either.

For all variables in `var`, `x[i]` specifies the starting value for the variable `var[i]`. Similarly, `dj[i]` specifies the starting reduced cost for variable `var[i]`. For all ranged constraints specified in `rng`, `slack[i]` specifies the starting slack value for `rng[i]`. Similarly, `pi[i]` specifies the starting dual value for `rng[i]`.

This information is exploited at the next call to `solve`, to construct a starting point for the algorithm, provided that the `AdvInd` parameter is set to a value greater than or equal to one. In particular, if the extracted model is an LP, and the root algorithm is dual or primal, the information is used to construct a starting basis for the simplex method for the original model, if `AdvInd` is set to 1 (one). If `AdvInd` is set to 2, the information is used to construct a starting basis for the presolved model.

If the extracted model is a MIP, only `x` values can be used, and a value must be specified for all variables of type `ILOINT`. If the provided values are compatible with an integer feasible solution, that solution becomes the incumbent for the next search; otherwise, the starting information is ignored. The parameter `IloCplex::MIPStart` must be turned on (that is, set to `IloTrue`) for the starting point to take effect.

```java
public IloBool solve(IloCplex::Goal goal)
```

This method initializes the goal stack of the root node with `goal` before starting the branch & cut search. The search tree will be defined by the execute method of `goal` and
public IloBool solve()

This method solves the model currently extracted to the invoking IloCplex object. The method returns IloTrue if it finds a solution (not necessarily an optimal one).

public IloBool solveFixed()

After the invoking algorithm has solved the extracted MIP model to a feasible (but not necessarily optimal) solution as a MIP, this member function solves the relaxation of the model obtained by fixing all the integer variables of the extracted MIP to the values of the current solution.

public IloCplex::Callback use(IloCplex::Callback cb)

This method instructs the invoking IloCplex object to use cb as a callback. If a callback of the same type as cb is already being used by the invoking IloCplex object, the previously used callback will be overridden. If the callback object cb is already being used by another IloCplex object, a copy of cb will be used instead. A handle to the callback that is installed in the invoking IloCplex object is returned. See IloCplex::Callback1 for a discussion of how to implement callbacks.

public void writeBasis(const char * name)

Writes the current Simplex basis to the file indicated by name. By convention, the file extension is .bas. The BAS file format is documented in the reference manual ILOG CPLEX File Formats.

public void writeMIPStart(const char * name)

Writes MIP start information to a file indicated by name. By convention, the file extension is .mst. The MST file format is documented in the reference manual ILOG CPLEX File Formats.

public void writeOrder(const char * filename)

This method writes a priority order to the file filename. If a priority order has been associated with the CPLEX problem object, or the parameter Mipordtype is nonzero, or a MIP feasible solution exists, this method writes the priority order into the indicated file. By convention, the file extension is .ord. The ORD file format is documented in the reference manual ILOG CPLEX File Formats.

public void writeParam(const char * name)

Writes the parameter name and its current setting into the file indicated by name for all the CPLEX parameters that are not currently set at their default.
By convention, the file extension is .prm. The PRM file format is documented in the reference manual *ILOG CPLEX File Formats*.

public void writeTextSolution(const char * name)

Writes a solution file in text format for the current problem to the file indicated by name.

public void writeTree(const char * name)

Writes the branch-and-bound tree from the last optimization to a file specified by name in TRE format.

By convention, the file extension is .tre. The TRE file format is documented in the reference manual *ILOG CPLEX File Formats*.

public void writeVectors(const char * name)

This method writes solution information from a barrier optimization (without crossover) into the file indicated by name. By convention, the file extension is .vec. The VEC file format is documented in the reference manual *ILOG CPLEX File Formats*.
IloCplex::Algorithm

Category Inner Enumeration

Definition File ilocplexi.h

Include Files ilocplex.h

Synopsis
Algorithm{
NoAlg,
AutoAlg,
Primal,
Dual,
Barrier,
Sifting,
Concurrent,
Network
};

Description The enumeration IloCplex::Algorithm lists the available algorithms in CPLEX to solve continuous models as controlled by the parameters IloCplex::RootAlg and IloCplex::NodeAlg.

See Also IloCplex, IloCplex::IntParam, getAlgorithm, getSubAlgorithm, RootAlg, NodeAlg

Fields
NoAlg = CPX_ALG_NONE
AutoAlg = CPX_ALG_AUTOMATIC
Primal = CPX_ALG_PRIMAL
Dual = CPX_ALG_DUAL
Barrier = CPX_ALG_BARRIER
Sifting = CPX_ALG_SIFTING
Concurrent = CPX_ALG_CONCURRENT
Network

= CPX_ALG_NET
IloCplex::BarrierCallbackI

**Category**
Inner Class

**Inheritance Path**

**Definition File**
ilocplexi.h

**Include Files**
ilcplex/ilocplex.h

---

### Constructor Summary

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>protected</td>
<td>BarrierCallbackI()</td>
</tr>
</tbody>
</table>

---

### Method Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>protected IloNum</td>
<td>getDualObjValue()</td>
</tr>
</tbody>
</table>

---

**Inherited methods from IloCplex::ContinuousCallbackI**

ContinuousCallbackI::getDualInfeasibility, ContinuousCallbackI::getInfeasibility, ContinuousCallbackI::getNiterations, ContinuousCallbackI::getObjValue, ContinuousCallbackI::isDualFeasible, ContinuousCallbackI::isFeasible

---

**Inherited methods from IloCplex::CallbackI**
An instance of the class IloCplex::BarrierCallbackI represents a user-written callback in an application that uses an instance of IloCplex to solve a problem by means of the barrier optimizer. IloCplex calls the user-written callback after each iteration during optimization with the barrier method. If an attempt is made to access information not available to an instance of this class, an exception is thrown.

The constructor and methods of this class are for use in deriving a user-written callback class and in implementing the main method there.

For more information about the barrier optimizer, see the ILOG CPLEX User's Manual.

See Also
ILOBARRIERCALLBACK0, IloCplex, IloCplex::Callback, IloCplex::CallbackI, IloCplex::ContinuousCallbackI

Constructors
protected BarrierCallbackI()

This constructor creates a callback for use in an application of the barrier optimizer.

Methods
protected IloNum getDualObjValue()

This method returns the current dual objective value of the solution in the instance of IloCplex at the time the invoking callback is executed.
IloCplex::BasisStatus

Category: Inner Enumeration

Definition File: ilocplexi.h

Include Files: ilcplex/ilocplex.h

Synopsis:

```cpp
BasisStatus{
    NotABasicStatus,
    Basic,
    AtLower,
    AtUpper,
    FreeOrSuperbasic
};
```

Description:
The enumeration IloCplex::BasisStatus lists values that the status of variables or range constraints may assume in a basis. NotABasicStatus is not a valid status for a variable. A basis containing such a status does not constitute a valid basis. The basis status of a ranged constraint corresponds to the basis status of the corresponding slack or artificial variable that IloCplex manages for it.

See Also:
IloCplex, IloCplex::BasisStatusArray

Fields:

- NotABasicStatus
- Basic
- AtLower
- AtUpper
- FreeOrSuperbasic
**IloCplex::BasisStatusArray**

**Category**  
Inner Type Definition

**Definition File**  
ilocplexi.h

**Include Files**  
ilocplex.h

**Synopsis**  
IloArray< BasisStatus > BasisStatusArray

**Description**  
This type defines an array-type for IloCplex::BasisStatus. The fully qualified name of a basis status array is IloCplex::BasisStatusArray.

**See Also**  
IloCplex, IloCplex::BasisStatus
**IloCplex::BoolParam**

**Category**  
Inner Enumeration

**Definition File**  
ilocplexi.h

**Include Files**  
ilcplex/ilocplex.h

**Synopsis**

```cpp
IloCplex::BoolParam {
    DepInd,
    PreInd,
    MIPStart,
    ReverseInd,
    XXXInd,
    MIPordInd,
    RelaxPreInd,
    PerInd,
    PreLinear,
    DataCheck,
    BarOOC,
    Symmetry,
    QPmakePSDInd,
    FinalFactor
};
```

**Description**

The enumeration `IloCplex::BoolParam` lists the parameters of CPLEX that require Boolean values. Boolean values are also known in certain contexts as binary values or as zero-one (0-1) values. Use these values with the methods that accept Boolean parameters: `IloCplex::getParam` and `IloCplex::setParam`.

See the reference manual *ILOG CPLEX Parameters* for more information about these parameters. Also see the user’s manual for examples of their use.

**See Also**

*IloCplex*

**Fields**

```cpp
DepInd
    = CPX_PARAM_DEPIND
PreInd
    = CPX_PARAM_PREIND
MIPStart
    = CPX_PARAM_MIPSTART
ReverseInd
    = CPX_PARAM_REVERSEIND
XXXInd
    = CPX_PARAM_XXXIND
```
= CPX_PARAM_XXXIND
MIPOrdInd
= CPX_PARAM_MIPORDIND
RelaxPreInd
= CPX_PARAM_RELAXPREIND
PerInd
= CPX_PARAM_PERIND
PreLinear
= CPX_PARAM_PRELINEAR
DataCheck
= CPX_PARAM_DATACHECK
BarOOC
= CPX_PARAM_BAROOC
Symmetry
= CPX_PARAM_SYMMETRY
QPmakePSDInd
= CPX_PARAM_QPMAKEPSDIND
FinalFactor
= CPX_PARAM_FINALFACTOR
IloCplex::BranchCallbackI

Category Inner Class

InheritancePath

Definition File ilocplexi.h

Include Files ilcplex/ilocplex.h

Constructor Summary

| protected BranchCallbackI() |

Method Summary

<p>| protected IloNum | getBranch(IloNumVarArray, IloNumArray, IloCplex::BranchDirectionArray, IloInt) |
| protected BranchCallbackI::BranchType | getBranchType() |
| protected IloInt | getNbranches() |
| protected NodeId | getNodeId() |
| protected IloBool | isIntegerFeasible() |
| protected NodeId | makeBranch(const IloConstraintArray, const IloIntVarArray, const IloNumArray, const IloCplex::BranchDirectionArray, IloNum, NodeData *) |</p>
<table>
<thead>
<tr>
<th>protected NodeId</th>
<th>makeBranch(const IloConstraintArray, const IloNumVarArray, const IloNumArray, const IloCplex::BranchDirectionArray, IloNum, NodeData *)</th>
</tr>
</thead>
<tbody>
<tr>
<td>protected NodeId</td>
<td>makeBranch(const IloConstraint, IloNum, NodeData *)</td>
</tr>
<tr>
<td>protected NodeId</td>
<td>makeBranch(const IloConstraintArray, IloNum, NodeData *)</td>
</tr>
<tr>
<td>protected NodeId</td>
<td>makeBranch(const IloIntVar, IloNum, IloCplex::BranchDirection, IloNum, NodeData *)</td>
</tr>
<tr>
<td>protected NodeId</td>
<td>makeBranch(const IloNumVar, IloNum, IloCplex::BranchDirection, IloNum, NodeData *)</td>
</tr>
<tr>
<td>protected NodeId</td>
<td>makeBranch(const IloIntVarArray, const IloNumArray, const IloCplex::BranchDirectionArray, IloNum, NodeData *)</td>
</tr>
<tr>
<td>protected NodeId</td>
<td>makeBranch(const IloNumVarArray, const IloNumArray, const IloCplex::BranchDirectionArray, IloNum, NodeData *)</td>
</tr>
<tr>
<td>protected void</td>
<td>prune()</td>
</tr>
</tbody>
</table>

Inherited methods from `IloCplex::ControlCallbackI`
### Inherited methods from **IloCplex::MIPCallbackI**

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIPCallbackI::getBestObjValue</td>
</tr>
<tr>
<td>MIPCallbackI::getCutoff</td>
</tr>
<tr>
<td>MIPCallbackI::getDirection</td>
</tr>
<tr>
<td>MIPCallbackI::getIncumbentObjValue</td>
</tr>
<tr>
<td>MIPCallbackI::getIncumbentValue</td>
</tr>
<tr>
<td>MIPCallbackI::getIncumbentValue</td>
</tr>
<tr>
<td>MIPCallbackI::getIncumbentValues</td>
</tr>
<tr>
<td>MIPCallbackI::getMyThreadNum</td>
</tr>
<tr>
<td>MIPCallbackI::getNcliques</td>
</tr>
<tr>
<td>MIPCallbackI::getNcovers</td>
</tr>
<tr>
<td>MIPCallbackI::getNdisjunctiveCuts</td>
</tr>
<tr>
<td>MIPCallbackI::getNflowCovers</td>
</tr>
<tr>
<td>MIPCallbackI::getNflowPaths</td>
</tr>
<tr>
<td>MIPCallbackI::getNfractionalCuts</td>
</tr>
<tr>
<td>MIPCallbackI::getNGUBcovers</td>
</tr>
<tr>
<td>MIPCallbackI::getNimpliedBounds</td>
</tr>
<tr>
<td>MIPCallbackI::getNiterations</td>
</tr>
<tr>
<td>MIPCallbackI::getNMIIRs</td>
</tr>
<tr>
<td>MIPCallbackI::getNnodes</td>
</tr>
<tr>
<td>MIPCallbackI::getNremainingNodes</td>
</tr>
<tr>
<td>MIPCallbackI::getObjCoef</td>
</tr>
<tr>
<td>MIPCallbackI::getObjCoefs</td>
</tr>
<tr>
<td>MIPCallbackI::getObjCoefsF</td>
</tr>
<tr>
<td>MIPCallbackI::getObjCoefsS</td>
</tr>
<tr>
<td>MIPCallbackI::getPriority</td>
</tr>
<tr>
<td>MIPCallbackI::getUserThreads</td>
</tr>
<tr>
<td>MIPCallbackI::hasIncumbent</td>
</tr>
</tbody>
</table>

### Inherited methods from **IloCplex::CallbackI**

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>CallbackI::abort</td>
</tr>
<tr>
<td>CallbackI::duplicateCallback</td>
</tr>
<tr>
<td>CallbackI::getEnv</td>
</tr>
<tr>
<td>CallbackI::getModel</td>
</tr>
<tr>
<td>CallbackI::getNcols</td>
</tr>
<tr>
<td>CallbackI::getNQCs</td>
</tr>
<tr>
<td>CallbackI::getNrows</td>
</tr>
<tr>
<td>CallbackI::main</td>
</tr>
</tbody>
</table>
Description

An instance of the class IloCplex::BranchCallbackI represents a user-written callback in an application that uses an instance of IloCplex to solve a mixed integer program (MIP). The user-written callback is called prior to branching at a node in the branch & cut tree during the optimization of a MIP. It allows you to query how the invoking instance of IloCplex is about to create subnodes at the current node and gives you the option to override the selection made by the invoking instance of IloCplex. You can create zero, one, or two branches.

◆ The method prune removes the current node from the search tree. No subnodes from the current node will be added to the search tree.

◆ The method makeBranch tells an instance of IloCplex how to create a subproblem. You may call this method zero, one, or two times in every invocation of the branch callback. If you call it once, it creates one node; if you call it twice, it creates two nodes (one node at each call).

◆ If you call neither IloCplex::BranchCallbackI::prune nor IloCplex::BranchCallbackI::makeBranch, the instance of IloCplex proceeds with its own selection.

◆ Calling both IloCplex::BranchCallbackI::prune and IloCplex::BranchCallbackI::makeBranch in one invocation of a branch callback is an error and results in unspecified behavior.

The methods of this class are for use in deriving a user-written callback class and in implementing the main method there.

If an attempt is made to access information not available to an instance of this class, an exception is thrown.

See Also
ILOBRANCHCALLBACK0, IloCplex::BranchDirection, IloCplex::Callback, IloCplex::CallbackI, IloCplex::MIPCallbackI, IloCplex::ControlCallbackI

Constructors

protected BranchCallbackI()

This constructor creates a branch callback, that is, a control callback for splitting a node into two branches.

Methods

protected IloNum getBranch(IloNumVarArray vars, IloNumArray bounds,
This method accesses branching information for the i-th branch that the invoking instance of \texttt{IloCplex} is about to create. The parameter \( i \) must be between 0 (zero) and \((\text{getNbranches} - 1)\); that is, it must be a valid index of a branch; normally, it will be zero or one.

A branch is normally defined by a set of variables and the bounds for these variables. Branches that are more complex cannot be queried. The return value is the node estimate for that branch.

- The parameter \( \texttt{vars} \) contains the variables for which new bounds will be set in the \( i \)-th branch.
- The parameter \( \texttt{bounds} \) contains the new bounds for the variables listed in \( \texttt{vars} \); that is, \( \texttt{bounds}[j] \) is the new bound for \( \texttt{vars}[j] \).
- The parameter \( \texttt{dirs} \) indicates the branching direction for the variables in \( \texttt{vars} \).

\[
\texttt{dir}[j] = \texttt{IloCplex::BranchUp} \\
\texttt{dir}[j] = \texttt{IloCplex::BranchDown}
\]

This method returns the type of branching \texttt{IloCplex} is going to do for the current node.

\texttt{protected \texttt{BranchCallbackI::BranchType getBranchType()}}

This method returns the number of branches \texttt{IloCplex} is going to create at the current node.

\texttt{protected \texttt{IloInt getNbranches()}}

Returns the \texttt{NodeId} of the current node.

\texttt{protected \texttt{IloBool isIntegerFeasible()}}

This method returns \texttt{IloTrue} if the solution of the current node is integer feasible.

\texttt{protected \texttt{NodeId makeBranch(const IloConstraintArray cons,} \\
\texttt{const IloIntVarArray vars,} \\
\texttt{const IloNumArray bounds,} \\
\texttt{const \texttt{IloCplex::BranchDirectionArray dirs,} \\
\texttt{IloNum objestimate,} \\
\texttt{NodeId objestimate,} \\
\texttt{NodeData * data)}}
This method offers the same facilities as the other methods
IloCplex::BranchCallbackI::makeBranch, but for a branch specified by a set of constraints and a set of variables.

protected NodeId makeBranch(const IloConstraintArray cons,
const IloNumVarArray vars,
const IloNumArray bounds,
const IloCplex::BranchDirectionArray dirs,
IloNum objestimate,
NodeData * data)

This method offers the same facilities as the other methods
IloCplex::BranchCallbackI::makeBranch, but for a branch specified by only one constraint as
IloCplex::BranchCallbackI::makeBranch does for a branch specified by a set of constraints.

protected NodeId makeBranch(const IloConstraint con,
IloNum objestimate,
NodeData * data)

This method overrides the branch chosen by an instance of IloCplex, by specifying a branch on constraints. A method named makeBranch can be called zero, one, or two times in every invocation of the branch callback. If you call it once, it creates one node; if you call it twice, it creates two nodes (one node at each call). If you call it more than twice, it throws an exception.

◆ The parameter cons specifies an array of constraints that are to be added for the subnode being created.

◆ The parameter objestimate provides an estimate of the resulting optimal objective value for the subnode specified by this branch. The invoking instance of IloCplex may use this estimate to select nodes to process. Providing a wrong estimate will not influence the correctness of the solution, but it may influence performance. Using the objective value of the current node is usually a safe choice.

◆ The parameter data allows you to add an object of type IloCplex::MIPCallbackI::NodeData to the node representing the branch created by the makeBranch call. Such data objects must be instances of a user-written subclass of IloCplex::MIPCallbackI::NodeData.

protected NodeId makeBranch(const IloIntVar var,
IloNum bound,
IloCplex::BranchDirection dir,
IloNum objestimate,
protected NodeId makeBranch(const IloIntVarArray vars,
const IloNumArray bounds,
const IloCplex::BranchDirectionArray dirs,
IloNum objestimate,
NodeData * data)

This method overrides the branch chosen by an instance of IloCplex. A method named makeBranch can be called zero, one, or two times in every invocation of the branch callback. If you call it once, it creates one node; if you call it twice, it creates two nodes (one node at each call). If you call it more than twice, it throws an exception.

Each call specifies a branch; in other words, it instructs the invoking IloCplex object how to create a subnode from the current node by specifying new, tighter bounds for a set of variables.

◆ The parameter vars contains the variables for which new bounds will be set in the branch.
◆ The parameter bounds contains the new bounds for the variables listed in vars; that is, bounds[j] is the new bound to be set for vars[j].
◆ The parameter dirs indicates the branching direction for the variables in vars. dir[j] == IloCplex::BranchUp means that bounds[j] specifies a lower bound for vars[j].
   dirs[j] == IloCplex::BranchDown means that bounds[j] specifies an upper bound for vars[j].
◆ The parameter objestimate provides an estimate of the resulting optimal objective value for the subnode specified by this branch. The invoking instance of IloCplex may use this estimate to select nodes to process. Providing a wrong estimate will not influence the correctness of the solution, but it may influence performance. Using the objective value of the current node is usually a safe choice.
The parameter `data` allows you to add an object of type `IloCplex::MIPCallbackI::NodeData` to the node representing the branch created by the `makeBranch` call. Such data objects must be instances of a user-written subclass of `IloCplex::MIPCallbackI::NodeData`.

```cpp
protected NodeId makeBranch(const IloNumVarArray vars,
                           const IloNumArray bounds,
                           const IloCplex::BranchDirectionArray dirs,
                           IloNum objestimate,
                           NodeData * data)
```

This method overrides the branch chosen by an instance of `IloCplex`. A method named `makeBranch` can be called zero, one, or two times in every invocation of the branch callback. If you call it once, it creates one node; if you call it twice, it creates two nodes (one node at each call). If you call it more than twice, it throws an exception.

Each call specifies a branch; in other words, it instructs the invoking `IloCplex` object how to create a subnode from the current node by specifying new, tighter bounds for a set of variables.

- The parameter `vars` contains the variables for which new bounds will be set in the branch.
- The parameter `bounds` contains the new bounds for the variables listed in `vars`; that is, `bounds[j]` is the new bound to be set for `vars[j]`.
- The parameter `dirs` indicates the branching direction for the variables in `vars`. `dir[j] == IloCplex::BranchUp` means that `bounds[j]` specifies a lower bound for `vars[j]`. `dirs[j] == IloCplex::BranchDown` means that `bounds[j]` specifies an upper bound for `vars[j]`.
- The parameter `objestimate` provides an estimate of the resulting optimal objective value for the subnode specified by this branch. The invoking instance of `IloCplex` may use this estimate to select nodes to process. Providing a wrong estimate will not influence the correctness of the solution, but it may influence performance. Using the objective value of the current node is usually a safe choice.
- The parameter `data` allows you to add an object of type `IloCplex::MIPCallbackI::NodeData` to the node representing the branch created by the `makeBranch` call. Such data objects must be instances of a user-written subclass of `IloCplex::MIPCallbackI::NodeData`.

```cpp
protected void prune()
```

By calling this method, you instruct the CPLEX branch & cut search not to create any child nodes from the current node, or, in other words, to discard nodes below the current node; it does not revisit the discarded nodes below the current node. In short, it creates
no branches. It is an error to call both prune and makeBranch in one invocation of a callback.
BranchCallbackI::BranchType

Category            Inner Enumeration
Definition File     ilocplexi.h
Include Files       ilcplex/ilocplex.h
Synopsis

```cpp
BranchType{
    BranchOnVariable,
    BranchOnSOS1,
    BranchOnSOS2,
    BranchOnAny,
    UserBranch
};
```

Description

IloCplex::BranchCallbackI::BranchType is an enumeration limited in scope to the class IloCplex::BranchCallbackI. This enumeration is used by the method IloCplex::BranchCallbackI::getBranchType to tell what kind of branch IloCplex is about to do:

- **BranchOnVariable** indicates branching on a single variable.
- **BranchOnAny** indicates multiple bound changes and constraints will be used for branching.
- **BranchOnSOS1** indicates branching on an SOS of type 1.
- **BranchOnSOS2** indicates branching on an SOS of type 2.

See Also

IloCplex::BranchCallbackI

Fields

```cpp
BranchOnVariable
    = CPX_TYPE_VAR

BranchOnSOS1
    = CPX_TYPE_SOS1

BranchOnSOS2
    = CPX_TYPE_SOS2

BranchOnAny
    = CPX_TYPE_ANY

UserBranch
```
IloCplex::BranchDirection

Category Inner Enumeration

Definition File ilocplex1.h

Include Files ilocplex/ilocplex.h

Synopsis

BranchDirection{
    BranchGlobal,
    BranchDown,
    BranchUp
};

Description The enumeration IloCplex::BranchDirection lists values that can be used for specifying branch directions either with the branch direction parameter IloCplex::BrDir or with the methods IloCplex::setDirection and IloCplex::setDirections. The branch direction specifies which direction to explore first after branching on one variable.

See the reference manual ILOG CPLEX Parameters and the ILOG CPLEX User’s Manual for more information about these parameters. Also see the user’s manual for examples of their use.

See Also IloCplex, IloCplex::BranchDirectionArray

Fields

BranchGlobal
    = CPX_BRANCH_GLOBAL

BranchDown
    = CPX_BRANCH_DOWN

BranchUp
    = CPX_BRANCH_UP
**IloCplex::BranchDirectionArray**

**Category**  
Inner Type Definition

**Definition File**  
ilocplexi.h

**Include Files**  
ilocplex.h

**Synopsis**  
IloArray< BranchDirection > BranchDirectionArray

**Description**  
This type defines an array-type for IloCplex::BranchDirection. The fully qualified name of a branch direction array is IloCplex::BranchDirectionArray.

**See Also**  
IloCplex, IloCplex::BranchDirection
IloCplex::Callback

Category  Inner Class

InheritancePath

IloCplex::Callback

Definition File  ilocplexi.h

Include Files  ilocplex/ilocplex.h

Constructor Summary

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>Callback(IloCplex::CallbackI *)</td>
</tr>
</tbody>
</table>

Method Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>end()</td>
</tr>
<tr>
<td>public IloCplex::CallbackI *</td>
<td>getImpl()</td>
</tr>
<tr>
<td>public Callback::Type</td>
<td>getType()</td>
</tr>
</tbody>
</table>

Inner Enumeration

<table>
<thead>
<tr>
<th>Enumeration</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Callback::Type</td>
<td></td>
</tr>
</tbody>
</table>

Description

This class is the handle class for all callback implementation classes available for IloCplex. Callback implementation classes are user-defined classes derived from a subclass of IloCplex::CallbackI.

See Also

IloCplex, IloCplex::CallbackI

Constructors

public Callback(IloCplex::CallbackI * impl)

This constructor creates a callback handle object and initializes it to the implementation object passed as the argument.
Methods

public void **end**()**

This method deletes the implementation object pointed at by the invoking handle and sets the pointer to 0.

public **IloCplex::CallbackI** * **getImpl**()**

This method returns a pointer to the implementation object of the invoking handle.

public **IloCplex::CallbackI** * **getType**()**

This method returns the type of the callback implementation object referenced by the invoking handle.
IloCplex::CallbackI

Category Inner Class

InheritancePath

Definition File ilocplexi.h

Include Files ilcplex/ilocplex.h

Method Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>protected void CallbackI::abort()</td>
<td>This method is called to abort the optimization</td>
</tr>
<tr>
<td>protected virtual CallbackI* duplicateCallback()</td>
<td>This method is used to duplicate the callback</td>
</tr>
<tr>
<td>protected IloEnv getEnv()</td>
<td>This method returns the environment</td>
</tr>
<tr>
<td>protected IloModel getModel()</td>
<td>This method returns the model</td>
</tr>
<tr>
<td>protected IloInt getNcols()</td>
<td>This method returns the number of columns</td>
</tr>
<tr>
<td>protected IloInt getNQCs()</td>
<td>This method returns the number of quadratic constraints</td>
</tr>
<tr>
<td>protected IloInt getNrows()</td>
<td>This method returns the number of rows</td>
</tr>
<tr>
<td>protected virtual void main()</td>
<td>This method is called at various points during</td>
</tr>
</tbody>
</table>

Description

This is the abstract base class for user-written callback classes. It provides their common application programming interface (API). Callbacks may be called repeatedly at various points during an optimization: for each place a callback is called, ILOG CPLEX provides a separate callback class (derived from this class). Such a callback class provides the specific API as a protected method to use for the particular implementation.
You do not create instances of this class; rather, you use one of its child classes to implement your own callback. In order to implement your user-written callbacks with an instance of IloCplex, you should follow these steps:

◆ Determine which kind of callback you want to write, and choose the appropriate class for it. The class hierarchy in Tree may give you some ideas here.
◆ Derive your own subclass, MyCallbackI, say, from the appropriate predefined callback class.
◆ In your subclass of the callback class, use the protected API defined in the base class to implement the main routine of your user-written callback. (All constructors of predefined callback classes are protected; they can be called only from user-written derived subclasses.)
◆ In your subclass, implement the method duplicateCallback.
◆ Write a function myCallback, say, that creates an instance of your implementation class in the Concert Technology environment and returns it as an IloCplex::Callback handle.
◆ Create an instance of your callback class and pass it to the member function use.

Note: Macros ILOXXXCALLBACKn (for n from 0 to 7) are available to facilitate steps 2 through 5, where XXX stands for the particular callback under construction and n stands for the number of parameters that the function written in step 5 is to receive in addition to the environment parameter.

You can use one instance of a callback with only one instance of IloCplex. When you use a callback with a second instance of IloCplex, a copy will be automatically created using the method duplicateCallback, and that copy will be used instead.

Also, an instance of IloCplex takes account of only one instance of a particular callback at any given time. That is, if you call use more than once with the same class of callback, the last call overrides any previous one. For example, you can use only one primal simplex callback at a time, or you can use only one network callback at a time; and so forth.

There are two varieties of callbacks:

◆ Query callbacks enable your application to retrieve information about the current solution in an instance of IloCplex. The information available depends on the algorithm (primal simplex, dual simplex, barrier, mixed integer, or network) that you are using. For example, a query callback can return the current objective value, the number of simplex iterations that have been completed, and other details. Query
callbacks can also be called from presolve, probing, fractional cuts, and disjunctive cuts.

- Control callbacks enable you to direct the search when you are solving a MIP in an instance of IloCplex. For example, control callbacks enable you to select the next node to process or to control the creation of subnodes (among other possibilities).

Existing extractables should never be modified within a callback. Temporary extractables, such as arrays, expressions, and range constraints, can be created and modified. Temporary extractables are often useful, for example, for computing cuts.

See Also

ILOBARRIERCALLBACK0, ILOBRANCHCALLBACK0, IloCplex,
IloCplex::BarrierCallbackI, IloCplex::BranchCallbackI,
IloCplex::Callback, IloCplex::ControlCallbackI,
IloCplex::CrossoverCallbackI, IloCplex::CutCallbackI,
IloCplex::DisjunctiveCutCallbackI,
IloCplex::SimplexCallbackI,
IloCplex::FractionalCutCallbackI,
IloCplex::HeuristicCallbackI, IloCplex::IncumbentCallbackI,
IloCplex::ContinuousCallbackI, IloCplex::MIPCallbackI,
IloCplex::NetworkCallbackI, IloCplex::NodeCallbackI,
IloCplex::PresolveCallbackI, IloCplex::ProbingCallbackI,
IloCplex::SolveCallbackI, ILOCROSSOVERCALLBACK0,
ILOCUTCALLBACK0, ILOBRANCHCALLBACK0,
ILODISJUNCTIVECUTCALLBACK0, ILOFRACTIONALCUTCALLBACK0,
ILOHEURISTICCALLBACK0, ILOINCUMBENTCALLBACK0,
ILOCONTINUOUSCALLBACK0, ILOMIPCALLBACK0, ILONETWORKCALLBACK0,
ILONODECALLBACK0, ILORESOLVECALLBACK0, ILOPROBINGCALLBACK0,
ILOSIMPLEXCALLBACK0, ILORESOLVECALLBACK0

Methods

protected void abort()

This method stops the current optimization.

protected virtual CallbackI * duplicateCallback()

This virtual method must be implemented to create a copy of the invoking callback object on the same environment. Typically the following implementation will work for a callback class called MyCallbackI:

```
IloCplex::CallbackI* MyCallbackI::duplicateCallback() const {
    return (new (getEnv()) MyCallbackI(*this));
}
```

This method is called by an IloCplex object in two cases:
◆ When `cplex.use(cb)` is called for a callback object `cb` that is already used by another instance of `IloCplex`, a copy of the implementation object of `cb` is created by calling `duplicateCallback` and used in its place. The method `use` will return a handle to that copy.

◆ When a parallel optimizer is used, `IloCplex` creates copies of every callback that it uses by calling `duplicateCallback`. One copy of a callback is created for each additional thread being used in the parallel optimizer. During the optimization, each thread will invoke the copy corresponding to the thread number. The methods provided by the callback APIs are guaranteed to be threadsafe. However, when accessing parameters passed to callbacks or members stored in a callback, it is up to the user to make sure of thread safety by synchronizing access or creating distinct copies of the data in the implementation of `duplicateCallback`.

**protected IloEnv getEnv()**

This method returns the environment belonging to the `IloCplex` object that invoked the method `main`.

**protected IloModel getModel()**

This method returns the model currently extracted for the instance of `IloCplex` where the invoking callback executed.

**protected IloInt getNcols()**

This method returns the number of columns in the model currently being optimized.

**protected IloInt getNQCs()**

This method returns the number of quadratic constraints in the model currently being optimized.

**protected IloInt getNrows()**

This method returns the number of rows in the model currently being optimized.

**protected virtual void main()**

This virtual method is to be implemented by the user in a derived callback class to define the functionality of the callback. When an instance of `IloCplex` uses a callback (an instance of `IloCplex::CallbackI` or one of its derived subclasses), `IloCplex` calls this virtual method `main` at the point during the optimization at which the callback is executed.
Callback::Type

Category          Inner Enumeration
Definition File   ilocplexi.h
Include Files     ilcplex/ilocplex.h
Synopsis

Type{
  Continuous,
  Presolve,
  Simplex,
  Barrier,
  Crossover,
  Network,
  MIP,
  Probing,
  FractionalCut,
  DisjunctiveCut,
  Branch,
  Cut,
  Node,
  Heuristic,
  Incumbent,
  Solve,
  FlowMIRCut,
  _Number
};

Description
This enumeration type is used to identify the type of a callback implementation object referred to by an IloCplex::Callback handle.

See Also
IloCplex::Callback

Fields
Continuous
Presolve
Simplex
Barrier
Crossover
Network
MIP
Probing
FractionalCut
DisjunctiveCut
Branch
Cut
Node
Heuristic
Incumbent
Solve
FlowMIRCut
_Number
IloCplex::ContinuousCallbackI

Category: Inner Class

Inheritance Path:

- IloCplex::CallbackI
  - IloCplex::ContinuousCallbackI
  - IloCplex::BarrierCallbackI
  - IloCplex::SimplexCallbackI

Definition File: ilocplexi.h

Include Files: ilcplex/ilocplex.h

Constructor Summary:

<table>
<thead>
<tr>
<th>Access</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>protected</td>
<td>ContinuousCallbackI()</td>
</tr>
</tbody>
</table>

Method Summary:

<table>
<thead>
<tr>
<th>Access</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>protected</td>
<td>IloNum ContinuousCallbackI::getDualInfeasibility()</td>
</tr>
<tr>
<td>protected</td>
<td>IloNum ContinuousCallbackI::getInfeasibility()</td>
</tr>
<tr>
<td>protected</td>
<td>IloInt ContinuousCallbackI::getNiterations()</td>
</tr>
<tr>
<td>protected</td>
<td>IloNum ContinuousCallbackI::getObjValue()</td>
</tr>
<tr>
<td>protected</td>
<td>IloBool ContinuousCallbackI::isDualFeasible()</td>
</tr>
<tr>
<td>protected</td>
<td>IloBool ContinuousCallbackI::isFeasible()</td>
</tr>
</tbody>
</table>

Inherited methods from IloCplex::CallbackI:

- CallbackI::abort
- CallbackI::duplicateCallback
- CallbackI::getEnv
- CallbackI::getModel
- CallbackI::getNcols
- CallbackI::getNQCs
- CallbackI::getNrows
- CallbackI::main
Description

An instance of a class derived from IloCplex::ContinuousCallbackI represents a user-written callback in an ILOG CPLEX application that uses an instance of IloCplex with the primal simplex, dual simplex, or barrier optimizer. IloCplex calls the user-written callback after each iteration during an optimization of a problem solved at a node. This class offers methods for use within the callbacks you write. In particular, there are methods in this class to access primal and dual feasibility, number of iterations, and objective value.

The methods of this class are protected for use in deriving a user-written callback class and in implementing the main method there.

If an attempt is made to access information not available to an instance of this class, an exception is thrown.

Note: There are special callbacks for simplex and barrier, that is, IloCplex::SimplexCallbackI and IloCplex::BarrierCallbackI, respectively. Using a continuous callback sets this callback in both of these algorithms. If a special callback was already set for one of these algorithms, (for example, simplex) it is replaced by the general continuous callback.

See Also

IloCplex, IloCplex::Callback, IloCplex::CallbackI, ILOCONTINUOUSCALLBACK0

Constructors

protected ContinuousCallbackI()

This constructor creates a callback for use in an application that solves continuous models.

Methods

protected IloNum getDualInfeasibility()

This method returns the current dual infeasibility measure of the solution in the instance of IloCplex at the time the invoking callback is executed.

protected IloNum getInfeasibility()

This method returns the current primal infeasibility measure of the solution in the instance of IloCplex at the time the invoking callback is executed.

protected IloInt getNiterations()

This method returns the number of iterations completed so far by an instance of IloCplex at the invoking callback is executed.

protected IloNum getObjValue()
This method returns the current objective value of the solution in the instance of IloCplex at the time the invoking callback is executed.

protected IloBool isDualFeasible()
This method returns IloTrue if the current solution is dual feasible.

protected IloBool isFeasible()
This method returns IloTrue if the current solution is primal feasible.
IloCplex::ControlCallbackI

Category          Inner Class
InheritancePath

Definition File  ilocplexi.h
Include Files  ilcplex/ilocplex.h

Method Summary

<table>
<thead>
<tr>
<th>Type</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>protected</td>
<td>IloNum ControlCallbackI::getDownPseudoCost(const IloIntVar)</td>
</tr>
<tr>
<td>protected</td>
<td>IloNum ControlCallbackI::getDownPseudoCost(const IloNumVar)</td>
</tr>
<tr>
<td>protected</td>
<td>void ControlCallbackI::getFeasibilities(const IloIntVarArray, const IloIntVarArray)</td>
</tr>
<tr>
<td>protected</td>
<td>void ControlCallbackI::getFeasibilities(const IloNumVarArray, const IloNumVarArray)</td>
</tr>
<tr>
<td>protected</td>
<td>ControlCallbackI::getFeasibility(const IloIntVar)</td>
</tr>
<tr>
<td>protected</td>
<td>ControlCallbackI::getFeasibility(const IloNumVar)</td>
</tr>
</tbody>
</table>
### Inherited methods from `IloCplex::MIPCallbackI`

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>protected <code>ControlCallbackI::IntegerFeasibility</code></td>
<td><code>ControlCallbackI::getFeasibility(const IloSOS2)</code></td>
</tr>
<tr>
<td>protected <code>ControlCallbackI::IntegerFeasibility</code></td>
<td><code>ControlCallbackI::getFeasibility(const IloSOS1)</code></td>
</tr>
<tr>
<td>protected <code>IloNum</code></td>
<td><code>ControlCallbackI::getLB(const IloIntVar)</code></td>
</tr>
<tr>
<td>protected <code>IloNum</code></td>
<td><code>ControlCallbackI::getLB(const IloNumVar)</code></td>
</tr>
<tr>
<td>protected <code>void</code></td>
<td><code>ControlCallbackI::getLBs(IloNumArray, const IloIntVarArray)</code></td>
</tr>
<tr>
<td>protected <code>void</code></td>
<td><code>ControlCallbackI::getLBs(IloNumArray, const IloNumVarArray)</code></td>
</tr>
<tr>
<td>protected <code>NodeData *</code></td>
<td><code>ControlCallbackI::getNodeData()</code></td>
</tr>
<tr>
<td>protected <code>IloNum</code></td>
<td><code>ControlCallbackI::getObjValue()</code></td>
</tr>
<tr>
<td>protected <code>IloNum</code></td>
<td><code>ControlCallbackI::getSlack(const IloRange)</code></td>
</tr>
<tr>
<td>protected <code>void</code></td>
<td><code>ControlCallbackI::getSlacks(IloNumArray, const IloRangeArray)</code></td>
</tr>
<tr>
<td>protected <code>IloNum</code></td>
<td><code>ControlCallbackI::getUB(const IloIntVar)</code></td>
</tr>
<tr>
<td>protected <code>IloNum</code></td>
<td><code>ControlCallbackI::getUB(const IloNumVar)</code></td>
</tr>
<tr>
<td>protected <code>void</code></td>
<td><code>ControlCallbackI::getUBs(IloNumArray, const IloIntVarArray)</code></td>
</tr>
<tr>
<td>protected <code>void</code></td>
<td><code>ControlCallbackI::getUBs(IloNumArray, const IloNumVarArray)</code></td>
</tr>
<tr>
<td>protected <code>IloNum</code></td>
<td><code>ControlCallbackI::getUpPseudoCost(const IloIntVar)</code></td>
</tr>
<tr>
<td>protected <code>IloNum</code></td>
<td><code>ControlCallbackI::getUpPseudoCost(const IloNumVar)</code></td>
</tr>
<tr>
<td>protected <code>IloNum</code></td>
<td><code>ControlCallbackI::getValue(const IloIntVar)</code></td>
</tr>
<tr>
<td>protected <code>IloNum</code></td>
<td><code>ControlCallbackI::getValue(const IloNumVar)</code></td>
</tr>
<tr>
<td>protected <code>IloNum</code></td>
<td><code>ControlCallbackI::getValue(const IloExprArg)</code></td>
</tr>
<tr>
<td>protected <code>void</code></td>
<td><code>ControlCallbackI::getValues(IloNumArray, const IloIntVarArray)</code></td>
</tr>
<tr>
<td>protected <code>void</code></td>
<td><code>ControlCallbackI::getValues(IloNumArray, const IloNumVarArray)</code></td>
</tr>
<tr>
<td>protected <code>IloBool</code></td>
<td><code>ControlCallbackI::isSOSFeasible(const IloSOS2)</code></td>
</tr>
<tr>
<td>protected <code>IloBool</code></td>
<td><code>ControlCallbackI::isSOSFeasible(const IloSOS1)</code></td>
</tr>
</tbody>
</table>

---

**Note:** This list includes inherited methods from `IloCplex::MIPCallbackI`. The methods listed are part of the `ControlCallbackI` class and are protected, meaning they are intended for use within the class or by derived classes. The `IloCplex::ControlCallbackI` class provides a callback mechanism for the ILOG CPLEX C++ API, allowing users to monitor the solving process and react to various events.
Inherited methods from `IloCplex::CallbackI`

- `IloCplex::CallbackI::abort`, `IloCplex::CallbackI::duplicateCallback`, `IloCplex::CallbackI::getEnv`, `IloCplex::CallbackI::getModel`, `IloCplex::CallbackI::getNcols`, `IloCplex::CallbackI::getNQCs`, `IloCplex::CallbackI::getNrows`, `IloCplex::CallbackI::main`

**Inner Enumeration**

- `IloCplex::ControlCallbackI::IntegerFeasibility`  

**Inner Class**

- `IloCplex::ControlCallbackI::PresolvedVariableException`  

**Inner Type Def**

- `IloCplex::ControlCallbackI::IntegerFeasibilityArray`
Description

This class defines the common application programming interface (API) for the following classes that allow you to control the MIP search:

◆ IloCplex::SolveCallbackI
◆ IloCplex::CutCallbackI
◆ IloCplex::HeuristicCallbackI
◆ IloCplex::BranchCallbackI

An instance of one of these classes represents a user-written callback that intervenes in the search for a solution at a given node in an application that uses an instance of IloCplex to solve a mixed integer program (MIP). Control callbacks are tied to a node. They are called at each node during IloCplex branch & cut search. The user never subclasses the IloCplex::ControlCallbackI class directly; it only defines the common interface of those listed callbacks.

In particular, SolveCallbackI is called before solving the node relaxation and optionally allows substitution of its solution. IloCplex does this by default. After the node relaxation has been solved, either by an instance of SolveCallbackI or by IloCplex, the other control callbacks are called in the following order:

◆ IloCplex::CutCallbackI
◆ IloCplex::HeuristicCallbackI
◆ IloCplex::BranchCallbackI

If the cut callback added new cuts to the node relaxation, the node relaxation will be solved again using the solve callback, if used. The same is true if IloCplex generated its own cuts.

The methods of this class are protected and its constructor is private; you cannot directly subclass this class; you must derive from its subclasses.

If an attempt is made to access information not available to an instance of this class, an exception is thrown.

See Also

IloCplex, IloCplex::Callback, IloCplex::CallbackI, ControlCallbackI::IntegerFeasibility, IntegerFeasibilityArray, IloCplex::MIPCallbackI

Methods

protected IloNum getDownPseudoCost(const IloIntVar var)

This method returns the current pseudo cost for branching downward on the variable var.

protected IloNum getDownPseudoCost(const IloNumVar var)
This method returns the current pseudo cost for branching downward on the variable var.

protected void getFeasibilities(IntegerFeasibilityArray stat, const IloIntVarArray var)

This method indicates whether each of the variables in the array vars is integer feasible, integer infeasible, or implied integer feasible by putting the indication in the corresponding element of the array stats.

protected void getFeasibilities(IntegerFeasibilityArray stat, const IloNumVarArray var)

This method indicates whether each of the variables in the array vars is integer feasible, integer infeasible, or implied integer feasible by putting the indication in the corresponding element of the array stats.

protected ControlCallbackI::IntegerFeasibility getFeasibility(const IloIntVar var)

This method indicates whether the variable var is integer feasible, integer infeasible, or implied integer feasible in the current node solution.

protected ControlCallbackI::IntegerFeasibility getFeasibility(const IloNumVar var)

This method indicates whether the variable var is integer feasible, integer infeasible, or implied integer feasible in the current node solution.

protected ControlCallbackI::IntegerFeasibility getFeasibility(const IloSOS2 sos)

This method indicates whether the Special Ordered Set sos is integer feasible, integer infeasible, or implied integer feasible in the current node solution.

protected ControlCallbackI::IntegerFeasibility getFeasibility(const IloSOS1 sos)

This method indicates whether the Special Ordered Set sos is integer feasible, integer infeasible, or implied integer feasible in the current node solution.

protected IloNum getLB(const IloIntVar var)

This method returns the lower bound of var at the current node. This bound is likely to be different from the bound in the original model because an instance of IloCplex tightens bounds when it branches from a node to its subnodes.

protected IloNum getLB(const IloNumVar var)

This method returns the lower bound of var at the current node. This bound is likely to be different from the bound in the original model because an instance of IloCplex tightens bounds when it branches from a node to its subnodes.

protected void getLBs(IloNumArray val, const IloIntVarArray vars)
For each element of the array \( \text{vars} \), this method puts the lower bound at the current node into the corresponding element of the array \( \text{vals} \). These bounds are likely to be different from the bounds in the original model because an instance of \texttt{IloCplex} tightens bounds when it branches from a node to its subnodes.

protected void \texttt{getLBs} (IloNumArray \texttt{val},
const IloNumVarArray \texttt{vars})

This method puts the lower bound at the current node of each element of the array \( \text{vars} \) into the corresponding element of the array \( \text{vals} \). These bounds are likely to be different from the bounds in the original model because an instance of \texttt{IloCplex} tightens bounds when it branches from a node to its subnodes.

protected void \texttt{getLBs} (IloNumArray \texttt{val},
const IloNumVarArray \texttt{vars})

This method retrieves the \texttt{NodeData} object that may have previously been assigned to the current node by the user with the method \texttt{IloCplex::BranchCallbackI::makeBranch}. If no data object has been assigned to the current node, 0 (zero) will be returned.

protected NodeData * \texttt{getNodeData} ()

This method returns the objective value of the solution of the relaxation at the current node.

protected IloNum \texttt{getObjValue} ()

This method returns the slack value for the constraint indicated by \( \texttt{rng} \) in the solution of the relaxation at the current node.

protected void \texttt{getSlacks} (IloNumArray \texttt{val},
const IloRangeArray \texttt{con})

For each of the constraints in the array of ranges \( \texttt{rngs} \), this method puts the slack value in the solution of the relaxation at the current node into the corresponding element of the array \( \text{vals} \).

protected IloNum \texttt{getUB} (const IloIntVar \texttt{var})

This method returns the upper bound of the variable \( \texttt{var} \) at the current node. This bound is likely to be different from the bound in the original model because an instance of \texttt{IloCplex} tightens bounds when it branches from a node to its subnodes.

protected IloNum \texttt{getUB} (const IloIntVar \texttt{var})

This method returns the upper bound of the variable \( \texttt{var} \) at the current node. This bound is likely to be different from the bound in the original model because an instance of \texttt{IloCplex} tightens bounds when it branches from a node to its subnodes.

protected void \texttt{getUBs} (IloNumArray \texttt{val},
const IloIntVarArray \texttt{vars})
For each element in the array `vars`, this method puts the upper bound at the current node into the corresponding element of the array `vals`. The bounds are those in the relaxation at the current node. These bounds are likely to be different from the bounds in the original model because an instance of `IloCplex` tightens bounds when it branches from a node to its subnodes.

```c++
protected void getUBs(IloNumArray val,
                      const IloNumVarArray vars)
```

This method returns the current pseudo cost for branching upward on the variable `var`.

```c++
protected IloNum getUpPseudoCost(const IloIntVar var)
```

This method returns the value of the variable `var` in the solution of the relaxation at the current node.

```c++
protected IloNum getValue(const IloIntVar var)
```

This method returns the value of the expression `expr` in the solution of the relaxation at the current node.

```c++
protected IloNum getValue(const IloExprArg expr)
```

For each variable in the array `vars`, this method puts the value in the solution of the relaxation at the current node into the corresponding element of the array `vals`.

```c++
protected void getValues(IloNumArray val,
                         const IloIntVarArray vars)
```

For each variable in the array `vars`, this method puts the value in the solution of the relaxation at the current node into the corresponding element of the array `vals`.

```c++
protected IloBool isSOSFeasible(const IloSOS2 sos)
```

This method returns `IloTrue` if the solution of the LP at the current node is SOS feasible for the special ordered set indicated in its argument. The SOS set passed as a
parameter to this method may be of type 2. See the *ILOG CPLEX User’s Manual* for more explanation of types of special ordered sets.

**protected IloBool isSOSFeasible(const IloSOS sos1)**

This method returns *IloTrue* if the solution of the LP at the current node is SOS feasible for the special ordered set indicated in its argument. The SOS set passed as a parameter to this method may be of type 1. See the *ILOG CPLEX User’s Manual* for more explanation about these types of special ordered sets.
ControlCallbackI::IntegerFeasibility

Category
Inner Enumeration

Definition File
ilocplexi.h

Include Files
ilcplex/ilocplex.h

Synopsis

```
IntegerFeasibility {
    ImpliedInfeasible,
    Feasible,
    Infeasible,
    ImpliedFeasible
};
```

Description
The enumeration IloCplex::ControlCallbackI::IntegerFeasibility is an enumeration limited in scope to the class IloCplex::ControlCallbackI. This enumeration is used by IloCplex::ControlCallbackI::getFeasibility to represent the integer feasibility of a variable or SOS in the current node solution:

- **Feasible** indicates the variable or SOS is integer feasible.
- **ImpliedFeasible** indicates the variable or SOS has been presolved out. It will be feasible when all other integer variables or SOS are integer feasible.
- **Infeasible** indicates the variable or SOS is integer infeasible.

See Also
IloCplex, ControlCallbackI::IntegerFeasibilityArray

Fields

```
ImpliedInfeasible
Feasible
    = CPX_INTEGER_FEASIBLE
Infeasible
    = CPX_INTEGER_INFEASIBLE
ImpliedFeasible
    = CPX_IMPLIED_INTEGER_FEASIBLE
```
**IloCplex::CplexStatus**

**Category**  
Inner Enumeration

**Definition File**  
ilocplex1.h

**Include Files**  
ilcplex/ilocplex.h

**Synopsis**  
```c++
CplexStatus {
  Unknown, 
  Optimal, 
  Unbounded, 
  Infeasible, 
  InfOrUnbd, 
  OptimalInFeas, 
  NumBest, 
  FeasibleRelaxed, 
  OptimalRelaxed, 
  AbortObjLim, 
  AbortPrimObjLim, 
  AbortDualObjLim, 
  AbortItLim, 
  AbortTimeLim, 
  AbortUser, 
  OptimalFaceUnbounded, 
  OptimalTo1, 
  SolLim, 
  NodeLimFeas, 
  NodeLimInfeas, 
  FailFeas, 
  FailInfeas, 
  MemLimFeas, 
  MemLimInfeas, 
  FailFeasNoTree, 
  FailInfeasNoTree
};
```

**Description**  
The enumeration `IloCplex::CplexStatus` lists values that the status of an `IloCplex` algorithm can assume. These values can be accessed via the method `getCplexStatus` and `getCplexSubStatus` and provide information about what the algorithm learned about the active model in the most recent invocation of the method `solve` or `feasOpt`. It may also tell why the algorithm terminated.

See the group `optim.cplex.solutionstatus` in the `Callable Library Reference Manual`, where they are listed in alphabetic order, or the topic `Interpreting Solution Status Codes` in the Overview of the APIs, where they are listed in numeric order, for more information about these values. Also see the `ILOG CPLEX User’s Manual` for examples of their use.
See also the enumeration IloAlgorithm::Status in the *ILOG Concert Technology Reference Manual*.

### Fields

- **Unknown**
  - `= CPX_STAT_UNKNOWN`
- **Optimal**
  - `= CPX_STAT_OPTIMAL`
- **Unbounded**
  - `= CPX_STAT_UNBOUNDED`
- **Infeasible**
  - `= CPX_STAT_INFEASIBLE`
- **InfOrUnbd**
  - `= CPX_STAT_INFOrUNBD`
- **OptimalInfeas**
  - `= CPX_STAT_OPTIMAL_INFEAS`
- **NumBest**
  - `= CPX_STAT_NUM_BEST`
- **FeasibleRelaxed**
  - `= CPX_STAT_FEASIBLE_RELAXED`
- **OptimalRelaxed**
  - `= CPX_STAT_OPTIMAL_RELAXED`
- **AbortObjLim**
  - `= CPX_STAT_ABORT_OBJ_LIM`
- **AbortPrimObjLim**
  - `= CPX_STAT_ABORT_PRIM_OBJ_LIM`
- **AbortDualObjLim**
  - `= CPX_STAT_ABORT_DUAL_OBJ_LIM`
- **AbortItLim**
  - `= CPX_STAT_ABORT_IT_LIM`
- **AbortTimeLim**
  - `= CPX_STAT_ABORT_TIME_LIM`
- **AbortUser**
  - `= CPX_STAT_ABORT_USER`
- **OptimalFaceUnbounded**
= CPX_STAT_OPTIMAL_FACE_UNBOUNDED
OptimalTol
= CPXMIP_OPTIMAL_TOL
SolLim
= CPXMIP_SOL_LIM
NodeLimFeas
= CPXMIP_NODE_LIM_FEAS
NodeLimInfeas
= CPXMIP_NODE_LIM_INFEAS
FailFeas
= CPXMIP_FAIL_FEAS
FailInfeas
= CPXMIP_FAIL_INFEAS
MemLimFeas
= CPXMIP_MEM_LIM_FEAS
MemLimInfeas
= CPXMIP_MEM_LIM_INFEAS
FailFeasNoTree
= CPXMIP_FAIL_FEAS_NO_TREE
FailInfeasNoTree
= CPXMIP_FAIL_INFEAS_NO_TREE
**IloCplex::CrossoverCallbackI**

**Category**
Inner Class

**InheritancePath**

```
IloCplex::CallbackI
   IloCplex::CrossoverCallbackI
```

**Definition File**
ilocplexi.h

**Include Files**
ilcplex/ilcplex.h

### Constructor Summary

```
protected CrossoverCallbackI()
```

### Method Summary

```
protected IloInt getNdualExchanges()
protected IloInt getNdualPushes()
protected IloInt getNprimalExchanges()
protected IloInt getNprimalPushes()
protected IloInt getNsuperbasics()
```

### Inherited methods from IloCplex::CallbackI

```
CallbackI::abort, CallbackI::duplicateCallback, CallbackI::getEnv,
CallbackI::getModel, CallbackI::getNcols, CallbackI::getNQCs,
CallbackI::getNrows, CallbackI::main
```

**Description**

An instance of the class `IloCplex::CrossoverCallbackI` represents a user-written callback in an application that uses an instance of `IloCplex` to solve a problem by means of the barrier optimizer with the crossover option. An instance of `IloCplex`
calls this callback regularly during crossover. For details about the crossover option, see the ILOG CPLEX User's Manual.

The constructor and methods of this class are protected for use in deriving a user-written callback class and in implementing the main method there.

If an attempt is made to access information not available to an instance of this class, an exception is thrown.

See Also

IloCplex, IloCplex::Callback, IloCplex::CallbackI, ILOCROSSOVERCALLBACK0

Constructors

protected CrossoverCallbackI()

This constructor creates a callback for use in an application with the crossover option of the barrier optimizer.

Methods

protected IloInt getNdualExchanges()

This method returns the number of dual exchange operations executed so far during crossover by the instance of IloCplex that executes the invoking callback.

protected IloInt getNdualPushes()

This method returns the number of dual push operations executed so far during crossover by the instance of IloCplex that executes the invoking callback.

protected IloInt getNprimalExchanges()

This method returns the number of primal exchange operations executed so far during crossover by the instance of IloCplex that executes the invoking callback.

protected IloInt getNprimalPushes()

This method returns the number of primal push operations executed so far during crossover by the instance of IloCplex that executes the invoking callback.

protected IloInt getNsuperbasics()

This method returns the number of super basics currently present in the basis being generated with crossover by the instance of IloCplex that executes the invoking callback.
**IloCplex::CutCallbackI**

**Category**
Inner Class

**Inheritance Path**

![Inheritance Diagram](image)

**Definition File**
ilocplexi.h

**Include Files**
ilcplex/ilocplex.h

**Constructor Summary**

| protected | CutCallbackI() |

**Method Summary**

| protected | IloConstraint | add(IloConstraint) |
| protected | IloConstraint | addLocal(IloConstraint) |

**Inherited methods from** IloCplex::ControlCallbackI
IloCplex::CutCallbackI

Description
An instance of the class IloCplex::CutCallbackI represents a user-written callback in an application that uses an instance of IloCplex to solve a mixed integer programming problem (a MIP). This class offers a method to add a local or global cut.
from a user-written callback. More than one cut can be added in this callback by calling the method `add` or `addLocal` multiple times. All added cuts must be linear.

The constructor and methods of this class are protected for use in deriving a user-written callback class and in implementing the `main` method there.

If an attempt is made to access information not available to an instance of this class, an exception is thrown.

See Also

IloCplex, IloCplex::Callback, IloCplex::CallbackI, IloCplex::MIPCallbackI, IloCplex::ControlCallbackI, ILOCUTCALLBACK0

Constructors

protected `CutCallbackI()`

This constructor creates a callback for use in an application with a user-defined cut to solve a MIP.

Methods

protected `IloConstraint add(IloConstraint con)`

This method adds a cut for the constraint indicated by `con`. This cut must be globally valid; it will not be removed by backtracking or any other means during the search. The added cut must be linear.

protected `IloConstraint addLocal(IloConstraint con)`

This method adds a local cut for the constraint indicated by `con`. IloCplex will manage the local cut in such a way that it will be active only when processing nodes of this subtree. The added cut must be linear.
**IloCplex::DeleteMode**

**Category**
Inner Enumeration

**Definition File**
ilocplexi.h

**Include Files**
ilcplex/ilocplex.h

**Synopsis**

```cpp
DeleteMode{
    LeaveBasis,
    FixBasis
};
```

**Description**

This enumeration lists the possible settings for the delete mode of IloCplex as controlled by the method `setDeleteMode` and queried by the method `getDeleteMode`.

- **IloCplex::LeaveBasis**
  
  With the default setting `IloCplex::LeaveBasis`, an existing basis will remain unchanged if variables or constraints are removed from the loaded LP model. This choice generally renders the basis unusable for a restart when CPLEX is solving the modified LP and the advance indicator (parameter `IloCplex::AdvInd`) is set to `IloTrue`.

- **IloCplex::FixBasis**
  
  In contrast, with delete mode set to `IloCplex::FixBasis`, the invoking object will do basis pivots in order to maintain a valid basis when variables or constraints are removed. This choice makes the delete operation more computation-intensive, but may give a better starting point for reoptimization after modification of the extracted model.

If no basis is present in the invoking object, the setting of the delete mode has no effect.

**See Also**

IloCplex

**Fields**

- LeaveBasis
- FixBasis
IloCplex::DisjunctiveCutCallbackI

Category Inner Class

InheritancePath

Definition File ilocplexi.h

Include Files ilcplex/ilocplex.h

**Constructor Summary**

| protected | DisjunctiveCutCallbackI() |

**Method Summary**

| protected IloNum | getProgress() |

**Inherited methods from IloCplex::MIPCallbackI**
Description
An instance of the class IloCplex::DisjunctiveCutCallbackI represents a user-written callback in an application that uses an instance of IloCplex to solve a mixed integer programming problem (a MIP). This class offers a method to check on the progress of the generation of disjunctive cuts.

The constructor and methods of this class are protected for use in deriving a user-written callback class and in implementing the main method there.

If an attempt is made to access information not available to an instance of this class, an exception is thrown.

See Also
IloCplex, IloCplex::Callback, IloCplex::CallbackI, IloCplex::MIPCallbackI

Constructors
protected DisjunctiveCutCallbackI()
This constructor creates a callback for use in an application where disjunctive cuts are generated.

Methods
protected IloNum getProgress()
This method returns the fraction of completion of the disjunctive cut generation pass.
IloCplex::DualPricing

Category       Inner Enumeration
Definition File ilocplexi.h
Include Files  ilcplex/ilocplex.h
Synopsis        DualPricing{
                  DPriIndAuto,
                  DPriIndFull,
                  DPriIndSteep,
                  DPriIndFullSteep,
                  DPriIndSteepQStart,
                  DPriIndDevex
                };

Description     The enumeration IloCplex::DualPricing lists values that the dual pricing parameter IloCplex:DPriInd can assume in IloCplex for use with the dual simplex algorithm. Use these values with the method IloCplex::setParam(IloCplex::DPriInd, value) when you set the dual pricing indicator.

                  See the reference manual ILOG CPLEX Parameters and the ILOG CPLEX User's Manual for more information about these parameters. Also see the user's manual for examples of their use.

See Also        IloCplex

Fields         DPriIndAuto
                  = CPX_DPRIIND_AUTO
DPriIndFull
                  = CPX_DPRIIND_FULL
DPriIndSteep
                  = CPX_DPRIIND_STEEP
DPriIndFullSteep
                  = CPX_DPRIIND_FULLSTEEP
DPriIndSteepQStart
                  = CPX_DPRIIND_STEEPQSTART
DPriIndDevex
                  = CPX_DPRIIND_DEVEEX
IloCplex::Exception

Category Inner Class

InheritancePath

Definition File ilocplexi.h

Include Files ilcplex/ilocplex.h

Method Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public IloInt getStatus()</td>
<td>The method returns the ILOG CPLEX error code of an exception thrown by a member of IloCplex. These error codes are detailed in the reference manual as the group optim.cplex.errorcodes. This method may also return negative values for subclasses of IloCplex::Exception, which are not listed in the reference manual. The exceptions listed in the reference manual are always thrown as instances of IloCplex::Exception and not as an instance of one of its derived classes.</td>
</tr>
</tbody>
</table>
IloCplex::FlowMIRCutCallbackI

Category Inner Class

InheritancePath

Definition File ilocplexi.h

Include Files ilocplex.h

Constructor Summary

| protected | FlowMIRCutCallbackI() |

Method Summary

| protected IloNum | getProgress() |

Inherited methods from IloCplex::MIPCallbackI
Description

An instance of the class IloCplex::FlowMIRCutCallback represents a user-written callback in an application that uses an instance of IloCplex to solve a mixed integer programming problem (a MIP). This class offers a member function to check on the progress of the generation of Flow MIR cuts.

The constructor and methods of this class are protected for use in deriving a user-written callback class and in implementing the main method there.

If an attempt is made to access information not available to an instance of this class, an exception is thrown.

See Also

IloCplex, IloCplex::Callback, IloCplex::CallbackI, IloCplex::MIPCallbackI

Constructors

protected FlowMIRCutCallbackI()

This constructor creates a callback for use in an application where flow MIR cuts are generated.

Methods

protected IloNum getProgress()

This method returns the fraction of completion of the cut generation pass for FlowMIR cuts.
IloCplex::FractionalCutCallbackI

Category: Inner Class

Inheritance Path:

Definition File: ilocplex.h
Include Files: ilcplex/ilocplex.h

Constructor Summary:

| protected | FractionalCutCallbackI() |

Method Summary:

| protected IloNum | getProgress() |

Inherited methods from IloCplex::MIPCallbackI
Description
An instance of the class IloCplex::FractionalCutCallbackI represents a user-written callback in an application that uses an instance of IloCplex to solve a mixed integer programming problem (a MIP). This class offers a method to check on the progress of the generation of fractional cuts.

The constructor and methods of this class are protected for use in deriving a user-written callback class and in implementing the main method there.

If an attempt is made to access information not available to an instance of this class, an exception is thrown.

See Also
IloCplex, IloCplex::Callback, IloCplex::CallbackI, IloCplex::MIPCallbackI

Constructors
protected FractionalCutCallbackI()
This constructor creates a callback for use in an application where fractional cuts are generated.

Methods
protected IloNum getProgress()
This method returns the fraction of completion of the fractional cut generation pass.

Inherited methods from IloCplex::CallbackI
CallbackI::abort, CallbackI::duplicateCallback, CallbackI::getEnv, CallbackI::getModel, CallbackI::getNcols, CallbackI::getNQCs, CallbackI::getNrows, CallbackI::main
IloCplex::Goal

Category Inner Class

InheritancePath

Definition File ilocplexi.h

Include Files ilcplex/ilocplex.h

Constructor Summary

public Goal(GoalBaseI *)
public Goal(const Goal &)
public Goal()
public Goal(IloConstraint)
public Goal(IloConstraintArray)

Method Summary

Goal operator=(const Goal &)

Description

Goals can be used to control the branch & cut search in IloCplex. Goals are implemented in the class IloCplex::GoalI. This is the handle class for CPLEX goals.

Goal objects are reference-counted. This means every instance of IloCplex::GoalI keeps track about how many handle objects refer to it. If this number drops to 0 the IloCplex::GoalI object is automatically deleted. As a consequence, whenever you deal with a goal, you must keep a handle object around rather than only a pointer to the implementation object. Otherwise, you risk ending up with a pointer to an implementation object that has already been deleted.


Constructors

public Goal(GoalBaseI * goalI)
Creates a new goal from a pointer to the implementation object.

public Goal(const Goal & goal)

This is the copy constructor of the goal.

public Goal()

Creates a 0 goal handle, that is, a goal with a 0 implementation object pointer. This is also referred to as an empty goal.

public Goal(IloConstraint cut)

Creates a new goal that will add the constraint cut as a local cut to the node where the goal is executed. As a local cut, the constraint will be active only in the subtree rooted at the node where the goal was executed. The lifetime of the constraint passed to a goal is tied to the lifetime of the Goal. That is, the constraint’s method end is called when the goal’s implementation object is deleted. As a consequence, the method end must not be called for constraints passed to this constructor explicitly.

public Goal(IloConstraintArray cut)

Creates a new goal that adds the constraints given in the array cut as local cuts to the node where the goal is executed. As local cuts, the constraints will be active only in the subtree rooted at the node where the goal was executed. The lifetime of the constraints and the array passed to a goal is tied to the lifetime of the Goal. That is, the constraint’s method end is called when the goal’s implementation object is deleted. As a consequence, method end must not be called for the constraints and the array passed to this constructor explicitly.

Methods

public Goal operator=(const Goal & goal)

This is the assignment operator. It increases the reference count of the implementation object of goal. If the invoking handle referred to an implementation object before the assignment operation, its reference count is decreased. If thereby the reference count becomes 0, the implementation object is deleted.
**IloCplex::GoalI**

**Category**  
Inner Class

**Inheritance Path**

**Definition File**  
ilocplexi.h

**Include Files**  
ilcplex/ilcplex.h

### Constructor Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>Goal(IloEnv)</td>
</tr>
</tbody>
</table>

### Method Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>abort()</td>
</tr>
<tr>
<td>public static</td>
<td>IloCplex::Goal AndGoal(IloCplex::Goal, IloCplex::Goal)</td>
</tr>
<tr>
<td>public static</td>
<td>IloCplex::Goal BranchAsCplexGoal(IloEnv)</td>
</tr>
<tr>
<td>public virtual</td>
<td>IloCplex::Goal duplicateGoal()</td>
</tr>
<tr>
<td>public virtual</td>
<td>IloCplex::Goal execute()</td>
</tr>
<tr>
<td>public static</td>
<td>IloCplex::Goal FailGoal(IloEnv)</td>
</tr>
<tr>
<td>public IloNum</td>
<td>getBestObjValue()</td>
</tr>
<tr>
<td>public IloNum</td>
<td>getBranch(IloNumVarArray, IloNumArray, IloCplex::BranchDirectionArray, IloInt)</td>
</tr>
<tr>
<td>public IloCplex::BranchType</td>
<td>getBranchType()</td>
</tr>
<tr>
<td>public IloNum</td>
<td>getCutoff()</td>
</tr>
<tr>
<td>public IloCplex::BranchDirection</td>
<td>getDirection(const IloIntVar)</td>
</tr>
<tr>
<td>public IloCplex::BranchDirection</td>
<td>getDirection(const IloNumVar)</td>
</tr>
<tr>
<td>public IloNum</td>
<td>getDownPseudoCost(const IloIntVar)</td>
</tr>
<tr>
<td>Public IloNum</td>
<td>getDownPseudoCost(const IloNumVar)</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>Public IloEnv</td>
<td>getEnv()</td>
</tr>
<tr>
<td>Public void</td>
<td>getFeasibilities(GoalI::IntegerFeasibilityArray, const IloIntVarArray)</td>
</tr>
<tr>
<td>Public void</td>
<td>getFeasibilities(GoalI::IntegerFeasibilityArray, const IloNumVarArray)</td>
</tr>
<tr>
<td>Public GoalI::IntegerFeasibility</td>
<td>getFeasibility(IloSOS2)</td>
</tr>
<tr>
<td>Public GoalI::IntegerFeasibility</td>
<td>getFeasibility(IloSOS1)</td>
</tr>
<tr>
<td>Public GoalI::IntegerFeasibility</td>
<td>getFeasibility(const IloIntVar)</td>
</tr>
<tr>
<td>Public GoalI::IntegerFeasibility</td>
<td>getFeasibility(const IloNumVar)</td>
</tr>
<tr>
<td>Public IloNum</td>
<td>getIncumbentObjValue()</td>
</tr>
<tr>
<td>Public IloNum</td>
<td>getIncumbentValue(const IloIntVar)</td>
</tr>
<tr>
<td>Public IloNum</td>
<td>getIncumbentValue(const IloNumVar)</td>
</tr>
<tr>
<td>Public void</td>
<td>getIncumbentValues(IloNumArray, const IloIntVarArray)</td>
</tr>
<tr>
<td>Public void</td>
<td>getIncumbentValues(IloNumArray, const IloNumVarArray)</td>
</tr>
<tr>
<td>Public IloNum</td>
<td>getLB(const IloIntVar)</td>
</tr>
<tr>
<td>Public IloNum</td>
<td>getLB(const IloNumVar)</td>
</tr>
<tr>
<td>Public void</td>
<td>getLBs(IloNumArray, const IloIntVarArray)</td>
</tr>
<tr>
<td>Public void</td>
<td>getLBs(IloNumArray, const IloNumVarArray)</td>
</tr>
<tr>
<td>Public IloModel</td>
<td>getModel()</td>
</tr>
<tr>
<td>Public IloInt</td>
<td>getMyThreadNum()</td>
</tr>
<tr>
<td>Public IloInt</td>
<td>getNBranches()</td>
</tr>
<tr>
<td>Public IloInt</td>
<td>getNCliques()</td>
</tr>
<tr>
<td>Public IloInt</td>
<td>getNCols()</td>
</tr>
<tr>
<td>Public IloInt</td>
<td>getNConstrCuts()</td>
</tr>
<tr>
<td>Public IloInt</td>
<td>getNFlowCovers()</td>
</tr>
<tr>
<td>Public IloInt</td>
<td>getNFlowPaths()</td>
</tr>
<tr>
<td>Public IloInt</td>
<td>getNFractionalCuts()</td>
</tr>
<tr>
<td>Public IloInt</td>
<td>getNGlobalCuts()</td>
</tr>
<tr>
<td>Public IloInt</td>
<td>getNMIRs()</td>
</tr>
<tr>
<td>Public IloInt</td>
<td>getNNodes()</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>public IloInt</td>
<td>getNremainingNodes()</td>
</tr>
<tr>
<td>public IloInt</td>
<td>getNrows()</td>
</tr>
<tr>
<td>public IloNum</td>
<td>getObjCoef(const IloIntVar)</td>
</tr>
<tr>
<td>public IloNum</td>
<td>getObjCoef(const IloNumVar)</td>
</tr>
<tr>
<td>public void</td>
<td>getObjCoefs(IloNumArray, const IloIntVarArray)</td>
</tr>
<tr>
<td>public void</td>
<td>getObjCoefs(IloNumArray, const IloNumVarArray)</td>
</tr>
<tr>
<td>public IloNum</td>
<td>getObjValue()</td>
</tr>
<tr>
<td>public IloNum</td>
<td>getPriority(const IloIntVar)</td>
</tr>
<tr>
<td>public IloNum</td>
<td>getPriority(const IloNumVar)</td>
</tr>
<tr>
<td>public IloNum</td>
<td>getSlack(const IloRange)</td>
</tr>
<tr>
<td>public void</td>
<td>getSlacks(IloNumArray, const IloRangeArray)</td>
</tr>
<tr>
<td>public IloNum</td>
<td>getUB(const IloIntVar)</td>
</tr>
<tr>
<td>public IloNum</td>
<td>getUB(const IloNumVar)</td>
</tr>
<tr>
<td>public void</td>
<td>getUBs(IloNumArray, const IloIntVarArray)</td>
</tr>
<tr>
<td>public void</td>
<td>getUBs(IloNumArray, const IloNumVarArray)</td>
</tr>
<tr>
<td>public IloNum</td>
<td>getUpPseudoCost(const IloIntVar)</td>
</tr>
<tr>
<td>public IloInt</td>
<td>getUserThreads()</td>
</tr>
<tr>
<td>public IloNum</td>
<td>getValue(const IloIntVar)</td>
</tr>
<tr>
<td>public IloNum</td>
<td>getValue(const IloNumVar)</td>
</tr>
<tr>
<td>public IloNum</td>
<td>getValue(const IloExpr)</td>
</tr>
<tr>
<td>public void</td>
<td>getValues(IloNumArray, const IloIntVarArray)</td>
</tr>
<tr>
<td>public void</td>
<td>getValues(IloNumArray, const IloNumVarArray)</td>
</tr>
<tr>
<td>public static IloCplex::Goal</td>
<td>GlobalCutGoal(IloConstraintArray)</td>
</tr>
<tr>
<td>public static IloCplex::Goal</td>
<td>GlobalCutGoal(IloConstraint)</td>
</tr>
<tr>
<td>public IloBool</td>
<td>hasIncumbent()</td>
</tr>
<tr>
<td>public IloBool</td>
<td>isIntegerFeasible()</td>
</tr>
<tr>
<td>public IloBool</td>
<td>isSOSFeasible(const IloSOS2)</td>
</tr>
<tr>
<td>public IloBool</td>
<td>isSOSFeasible(const IloSOS1)</td>
</tr>
<tr>
<td>public static IloCplex::Goal</td>
<td>OrGoal(IloCplex::Goal, IloCplex::Goal)</td>
</tr>
<tr>
<td>public static IloCplex::Goal</td>
<td>SolutionGoal(const IloIntVarArray, const IloNumArray)</td>
</tr>
<tr>
<td>public static IloCplex::Goal</td>
<td>SolutionGoal(const IloNumVarArray, const IloNumArray)</td>
</tr>
</tbody>
</table>
Goals can be used to control the branch & cut search in IloCplex. Goals are implemented in subclasses of the class IloCplex::GoalI. This is the base class for user-written implementation classes of CPLEX goals.

To implement your own goal you need to create a subclass of IloCplex::GoalI and implement its pure virtual methods execute and duplicateGoal. You may use one of the ILOCPLEXGOAL0 macros to assist you in doing so. After implementing your goal class, you use it by passing it to the solve method when solving the model.

The method duplicateGoal may be called by IloCplex to create copies of a goal when needed for parallel branch & cut search. Thus the implementation of this method must create and return an exact copy of the invoked object itself.

The method execute controls the branch & cut search of IloCplex by the goal it returns. When IloCplex processes a node, it pops the top goal from the node's goal stack and calls method execute of that goal. It continues executing the top goal from the stack until the node is deactivated or the goal stack is empty. If the goal stack is empty, IloCplex proceeds with the built-in search strategy for the subtree rooted at the current node.

The class IloCplex::GoalI provides several methods for querying information about the current node. The method execute controls how to proceed with the branch & cut search via the goal it returns. The returned goal, unless it is the 0 goal, is pushed on the goal stack and will thus be executed next.

See also the chapter about goals in the ILOG CPLEX User's Manual.

**Constructors**

```cpp
public GoalI(IloEnv env)
```

The goal constructor. It requires an instance of the same IloEnv as the IloCplex object with which to use the goal. The environment can later be queried by calling method getEnv.
Methods

public void abort()

Abort the optimization, that is, the execution of method solve currently in process.

public static IloCplex::Goal AndGoal(IloCplex::Goal goal1, IloCplex::Goal goal2)

The static methods AndGoal all return a goal that pushes the goals passed as parameters onto the goal stack in reverse order. As a consequence, the goals will be executed in the order they are passed as parameters to the AndGoal function.

public static IloCplex::Goal BranchAsCplexGoal(IloEnv env)

This static function returns a goal that creates the same branches as the currently selected built-in branch strategy of IloCplex would choose at the current node. This goal allows you to proceed with the IloCplex search strategy, but keeps the search under goal control, thereby giving you the option to intervene at any point.

This goal is also important when you use node evaluators while you use a built-in branching strategy.

For example, consider the execute method of a goal starting like this:

```cpp
if (!isIntegerFeasible())
    return AndGoal(BranchAsCplexGoal(getEnv()), this);
```

It would do something only when IloCplex found a solution it considers to be a candidate for a new incumbent. Note the test of integer feasibility before returning the BranchAsCplexGoal. Without it, BranchAsCplex would be executed for a solution IloCplex considers to be feasible, but IloCplex would not know how to branch on it. An endless loop would result.

public virtual IloCplex::Goal duplicateGoal()

This virtual method must be implemented by the user. It must return a copy of the invoking goal object. This method may be called by IloCplex when doing parallel branch & cut search.

public virtual IloCplex::Goal execute()

This virtual method must be implemented by the user to specify the logic of the goal. The instance of IloCplex::Goal returned by this method will be added to the goal stack of the node where the invoking goal is being executed for further execution.

public static IloCplex::Goal FailGoal(IloEnv env)

This static method creates a goal that fails. That means that the branch where the goal is executed will be pruned or, equivalently, the search is discontinued at that node and the node is discarded.
public IloNum getBestObjValue()

This method returns the currently best known bound on the optimal solution value of the problem at the time the invoking goal is executed by an instance of IloCplex while solving a MIP. When a model has been solved to optimality, this value matches the optimal solution value. Otherwise, this value is computed for a minimization (maximization) problem as the minimum (maximum) objective function value of all remaining unexplored nodes.

public IloNum getBranch(IloNumVarArray vars,
                        IloNumArray bounds,
                        IloCplex::BranchDirectionArray dirs,
                        IloInt i)

This method accesses branching information for the i-th branch that the invoking instance of IloCplex is about to create. The parameter i must be between 0 (zero) and getNbranches - 1; that is, it must be a valid index of a branch; normally, it will be zero or one.

A branch is normally defined by a set of variables and the bounds for these variables. Branches that are more complex cannot be queried. The return value is the node estimate for that branch.

◆ The parameter vars contains the variables for which new bounds will be set in the i-th branch.
◆ The parameter bounds contains the new bounds for the variables listed in vars; that is, bounds[j] is the new bound for vars[j].
◆ The parameter dirs indicates the branching direction for the variables in vars.

dir[j] == IloCplex::BranchUp
means that bounds[j] specifies a lower bound for vars[j].
dirs[j] == IloCplex::BranchDown
means that bounds[j] specifies an upper bound for vars[j].

public GoalI::BranchType getBranchType()

This method returns the type of branching IloCplex is going to do for the current node.

public IloNum getCutoff()

The method returns the current cutoff value. An instance of IloCplex uses the cutoff value (the value of the objective function of the subproblem at a node in the search tree) to decide when to prune nodes from the search tree (that is, when to cut off that node and discard the nodes beyond it). The cutoff value is updated whenever a new incumbent is found.

public IloCplex::BranchDirection getDirection(const IloIntVar var)
Returns the solution value of \( var \) in the incumbent solution at the time the invoking goal is executed by an instance of IloCplex while solving a MIP. If there is no incumbent, this method throws an exception.

```java
public IloCplex::BranchDirection getDirection(const IloNumVar var)
```

Returns the solution value of \( var \) in the incumbent solution at the time the invoking goal is executed by an instance of IloCplex while solving a MIP. If there is no incumbent, this method throws an exception.

```java
public IloNum getDownPseudoCost(const IloIntVar var)
```

This method returns the current pseudo cost for branching downward on the variable \( var \).

```java
public IloNum getDownPseudoCost(const IloNumVar var)
```

This method returns the current pseudo cost for branching downward on the variable \( var \).

```java
public IloEnv getEnv()
```

Returns the instance of IloEnv passed to the constructor of the goal.

```java
public void getFeasibilities(GoalI::IntegerFeasibilityArray stats, const IloIntVarArray vars)
```

This method considers whether each of the variables in the array \( vars \) is integer feasible, integer infeasible, or implied integer feasible and puts the indication in the corresponding element of the array \( stats \).

```java
public void getFeasibilities(GoalI::IntegerFeasibilityArray stats, const IloNumVarArray vars)
```

This method considers whether each of the variables in the array \( vars \) is integer feasible, integer infeasible, or implied integer feasible and puts the indication in the corresponding element of the array \( stats \).

```java
public GoalI::IntegerFeasibility getFeasibility(const IloSOS2 sos)
```

This method indicates whether the SOS \( sos \) is integer feasible, integer infeasible, or implied integer feasible in the current node solution.

```java
public GoalI::IntegerFeasibility getFeasibility(const IloSOS1 sos)
```

This method indicates whether the SOS \( sos \) is integer feasible, integer infeasible, or implied integer feasible in the current node solution.

```java
public GoalI::IntegerFeasibility getFeasibility(const IloIntVar var)
```

This method indicates whether the variable \( var \) is integer feasible, integer infeasible, or implied integer feasible in the current node solution.
This method indicates whether the variable \( \text{var} \) is integer feasible, integer infeasible, or implied integer feasible in the current node solution.

```java
public IloNum getIncumbentObjValue()
```

This method returns the value of the objective function of the incumbent solution (that is, the best integer solution found so far). If there is no incumbent, this method throws an exception.

```java
public IloNum getIncumbentValue(const IloIntVar var)
```

This method returns the value of \( \text{var} \) in the incumbent solution. If there is no incumbent, this method throws an exception.

```java
public IloNum getIncumbentValue(const IloNumVar var)
```

This method returns the value of \( \text{var} \) in the incumbent solution. If there is no incumbent, this method throws an exception.

```java
public void getIncumbentValues(IloNumArray val,
const IloIntVarArray vars)
```

Returns the value of each variable in the array \( \text{vars} \) with respect to the current incumbent solution, and it puts those values into the corresponding array \( \text{vals} \). If there is no incumbent, this method throws an exception.

```java
public void getIncumbentValues(IloNumArray val,
const IloNumVarArray vars)
```

Returns the value of each variable in the array \( \text{vars} \) with respect to the current incumbent solution, and it puts those values into the corresponding array \( \text{vals} \). If there is no incumbent, this method throws an exception.

```java
public IloNum getLB(const IloIntVar var)
```

This method returns the lower bound of \( \text{var} \) in the current node relaxation. This bound is likely to be different from the bound in the original model because an instance of \text{IloCplex} tightens bounds when it branches from a node to its subnodes.

```java
public IloNum getLB(const IloNumVar var)
```

This method returns the lower bound of \( \text{var} \) in the current node relaxation. This bound is likely to be different from the bound in the original model because an instance of \text{IloCplex} tightens bounds when it branches from a node to its subnodes.

```java
public void getLBS(IloNumArray vals,
const IloIntVarArray vars)
```

This method puts the lower bound in the current node relaxation of each element of the array \( \text{vars} \) into the corresponding element of the array \( \text{vals} \). These bounds are likely to be different from the bounds in the original model because an instance of \text{IloCplex} tightens bounds when it branches from a node to its subnodes.
This method puts the lower bound in the current node relaxation of each element of the array `vars` into the corresponding element of the array `vals`. These bounds are likely to be different from the bounds in the original model because an instance of `IloCplex` tightens bounds when it branches from a node to its subnodes.

```java
public IloModel getModel()
```

This method returns the model currently extracted for the instance of `IloCplex` where the invoking goal applies.

```java
public IloInt getMyThreadNum()
```

Returns the identifier of the parallel thread being currently executed. This number is between 0 (zero) and the value returned by the method `getUserThreads() - 1`.

```java
public IloInt getNbranches()
```

This method returns the number of branches `IloCplex` is going to create at the current node.

```java
public IloInt getNcliques()
```

Returns the total number of clique cuts that have been added to the model so far during the current optimization.

```java
public IloInt getNcols()
```

This method returns the number of columns in the current optimization model.

```java
public IloInt getNcovers()
```

Returns the total number of cover cuts that have been added to the model so far during the current optimization.

```java
public IloInt getNdisjunctiveCuts()
```

Returns the total number of disjunctive cuts that have been added to the model so far during the current optimization.

```java
public IloInt getNflowCovers()
```

Returns the total number of flow cover cuts that have been added to the model so far during the current optimization.

```java
public IloInt getNflowPaths()
```

Returns the total number of flow path cuts that have been added to the model so far during the current optimization.

```java
public IloInt getNfractionalCuts()
```

Returns the total number of fractional cuts that have been added to the model so far during the current optimization.

```java
public IloInt getNGUBcovers()
```

Returns the total number of GUB cover cuts that have been added to the model so far during the current optimization.
Returns the total number of GUB cover cuts that have been added to the model so far during the current optimization.

public IloInt getNimpliedBounds()

Returns the total number of implied bound cuts that have been added to the model so far during the current optimization.

public IloInt getNiterations()

Returns the total number of iterations executed so far during the current optimization to solve the node relaxations.

public IloInt getNMIRs()

Returns the total number of MIR cuts that have been added to the model so far during the current optimization.

public IloInt getNnodes()

This method returns the number of nodes already processed in the current optimization.

public IloInt getNremainingNodes()

This method returns the number of nodes left to explore in the current optimization.

public IloInt getNrows()

This method returns the number of rows in the current optimization model.

public IloNum getObjCoef(const IloIntVar var)

Returns the linear objective coefficient for var in the model currently being solved.

public IloNum getObjCoef(const IloNumVar var)

Returns the linear objective coefficient for var in the model currently being solved.

public void getObjCoefs(IloNumArray vals, const IloIntVarArray vars)

This method puts the linear objective coefficient of each of the variables in the array vars into the corresponding element of the array vals.

public void getObjCoefs(IloNumArray vals, const IloNumVarArray vars)

This method puts the linear objective coefficient of each of the variables in the array vars into the corresponding element of the array vals.

public IloNum getObjValue()

This method returns the objective value of the solution of the current node.

public IloNum getPriority(const IloIntVar var)

Returns the branch priority used for variable var in the current optimization.

public IloNum getPriority(const IloNumVar var)
Returns the branch priority used for variable var in the current optimization.

```java
public IloNum getSlack(const IloRange rng)
```

This method returns the slack value for the constraint indicated by rng in the solution of the current node relaxation.

```java
public void getSlacks(IloNumArray vals,
                     const IloRangeArray rngs)
```

This method puts the slack value in the solution of the current node relaxation of each of the constraints in the array of ranges rngs into the corresponding element of the array vals.

```java
public IloNum getUB(const IloIntVar var)
```

This method returns the upper bound of the variable var in the current node relaxation. This bound is likely to be different from the bound in the original model because an instance of IloCplex tightens bounds when it branches from a node to its subnodes.

```java
public IloNum getUB(const IloNumVar var)
```

This method returns the upper bound of the variable var in the current node relaxation. This bound is likely to be different from the bound in the original model because an instance of IloCplex tightens bounds when it branches from a node to its subnodes.

```java
public void getUBs(IloNumArray vals,
                   const IloIntVarArray vars)
```

This method puts the upper bound in the current node relaxation of each element of the array vars into the corresponding element of the array vals. These bounds are likely to be different from the bounds in the original model because an instance of IloCplex tightens bounds when it branches from a node to its subnodes.

```java
public void getUBs(IloNumArray vals,
                   const IloNumVarArray vars)
```

This method puts the upper bound in the current node relaxation of each element of the array vars into the corresponding element of the array vals. These bounds are likely to be different from the bounds in the original model because an instance of IloCplex tightens bounds when it branches from a node to its subnodes.

```java
public IloNum getUpPseudoCost(const IloIntVar var)
```

This method returns the current pseudo cost for branching upward on the variable var.

```java
public IloNum getUpPseudoCost(const IloNumVar var)
```

This method returns the current pseudo cost for branching upward on the variable var.

```java
public IloInt getUserThreads()
```

This method returns the total number of parallel threads currently running.

```java
public IloNum getValue(const IloIntVar var)
```
This method returns the value of the variable `var` in the solution of the current node relaxation.

```java
public IloNum getValue(const IloNumVar var)
```

This method returns the value of the variable `var` in the solution of the current node relaxation.

```java
public IloNum getValue(const IloExpr expr)
```

This method returns the value of the expression `expr` in the solution of the current node relaxation.

```java
public void getValues(IloNumArray vals, const IloIntVarArray vars)
```

This method puts the current node relaxation solution value of each variable in the array `vars` into the corresponding element of the array `vals`.

```java
public void getValues(IloNumArray vals, const IloNumVarArray vars)
```

This method puts the current node relaxation solution value of each variable in the array `vars` into the corresponding element of the array `vals`.

```java
public static IloCplex::Goal GlobalCutGoal(IloConstraintArray con)
```

This method creates a goal that when executed adds the constraints (provided in the parameter array `con`) as global cuts to the model. These global cuts must be valid for the entire model, not only for the current subtree. In other words, these global cuts will be respected at every node.

IloCplex takes over memory management for the cuts passed to the method `GlobalCutGoal`. Thus IloCplex will call the method `end` as soon as it can be discarded after the goal executes. Calling `end` yourself or the constraints in the array `con` passed to method `GlobalCutGoal` or the array itself is an error and must be avoided.

```java
public static IloCplex::Goal GlobalCutGoal(IloConstraint con)
```

This method creates a goal that when executed adds the constraint `con` (provided as a parameter) as global cuts to the model. These global cuts must be valid for the entire model, not only for the current subtree. In other words, these global cuts will be respected at every node.

IloCplex takes over memory management for the cut passed to the method `GlobalCutGoal`. Thus IloCplex will call the method `end` as soon as it can be discarded after the goal executes. Calling `end` yourself for the constraint passed to method `GlobalCutGoal` is an error and must be avoided.

```java
public IloBool hasIncumbent()
```

This method returns `IloTrue` if an integer feasible solution has been found.
public IloBool isIntegerFeasible()

This method returns IloTrue if the solution of the current node is integer feasible.

public IloBool isSOSFeasible(const IloSOS2 sos2)

This method returns IloTrue if the solution of the current node is SOS feasible for the special ordered set indicated in its argument. The SOS passed as a parameter to this method must be of type 2; the equivalent method for an SOS of type 1 is also available. See the User’s Manual for more about these types of special ordered sets.

public IloBool isSOSFeasible(const IloSOS1 sos1)

This method returns IloTrue if the solution of the current node is SOS feasible for the special ordered set indicated in its argument. The SOS passed as a parameter to this method must be of type 1; the equivalent method for an SOS of type 2 is also available. See the User’s Manual for more about these types of special ordered sets.

public static IloCplex::Goal OrGoal(IloCplex::Goal goal1,
                                       IloCplex::Goal goal2)

The static methods OrGoal all return a goal that creates as many branches (or, equivalently, subproblems) as there are parameters. Each of the subnodes will be initialized with the remaining goal stack of the current node. In addition, the goal parameter will be pushed on the goal stack of the corresponding subgoal. If more than six branches need to be created, instances of OrGoal can be combined.

public static IloCplex::Goal SolutionGoal(const IloIntVarArray vars,
                                       const IloNumArray vals)

This static method creates and returns a goal that attempts to inject a solution specified by setting the variables listed in array vars to the corresponding values listed in the array vals.

IloCplex will not blindly accept such a solution as a new incumbent. Instead, it will make sure that this solution is compatible with both the model and the goals. When checking feasibility with goals, it checks feasibility with both the goals that have already been executed and the goals that are still on the goal stack. Thus, in particular, IloCplex will reject any solution that is not compatible with the branching that has been done so far.

IloCplex takes over memory management for arrays vars and vals passed to SolutionGoal. Thus IloCplex will call method end for these arrays as soon as they can be discarded. Calling end for the arrays passed to SolutionGoal is an error and must be avoided.

public static IloCplex::Goal SolutionGoal(const IloNumVarArray vars,
                                       const IloNumArray vals)

This static method creates and returns a goal that attempts to inject a solution specified by setting the variables listed in array vars to the corresponding values listed in the array vals.
IloCplex will not blindly accept such a solution as a new incumbent. Instead, it will make sure that this solution is compatible with both the model and the goals. When checking feasibility with goals, it checks feasibility with both the goals that have already been executed and the goals that are still on the goal stack. Thus, in particular, IloCplex will reject any solution that is not compatible with the branching that has been done so far.

IloCplex takes over memory management for arrays vars and vals passed to SolutionGoal. Thus IloCplex will call method end for these arrays as soon as they can be discarded. Calling end for the arrays passed to SolutionGoal is an error and must be avoided.
**GoalI::BranchType**

**Category**
Inner Enumeration

**Definition File**
ilocplexi.h

**Include Files**
ilcplex/ilcplex.h

**Synopsis**

```cpp
BranchType {
    BranchOnVariable,
    BranchOnSOS1,
    BranchOnSOS2,
    BranchOnAny,
    UserBranch
};
```

**Description**

`IloCplex::GoalI::BranchType` is an enumeration limited in scope to the class `IloCplex::GoalI`. This enumeration is used by the method `IloCplex::GoalI::getBranchType` to tell what kind of branch `IloCplex` is about to make:

- `BranchOnVariable` indicates branching on a single variable.
- `BranchOnAny` indicates multiple bound changes and constraints will be used for branching.
- `BranchOnSOS1` indicates branching on an SOS of type 1.
- `BranchOnSOS2` indicates branching on an SOS of type 2.

**See Also**

`IloCplex::GoalI`

**Fields**

- `BranchOnVariable` = CPX_TYPE_VAR
- `BranchOnSOS1` = CPX_TYPE_SOS1
- `BranchOnSOS2` = CPX_TYPE_SOS2
- `BranchOnAny` = CPX_TYPE_ANY
- `UserBranch`
GoalI::IntegerFeasibility

Category          Inner Enumeration
Definition File   ilocplexi.h
Include Files     ilcplex/ilocplex.h
Synopsis

    IntegerFeasibility{
        ImpliedInfeasible,
        Feasible,
        Infeasible,
        ImpliedFeasible
    };

Description

The enumeration IloCplex::GoalI::IntegerFeasibility is an enumeration limited in scope to the class IloCplex::GoalI. This enumeration is used by IloCplex::GoalI::getFeasibility to access the integer feasibility of a variable or SOS in the current node solution:

◆ Feasible indicates the variable or SOS is integer feasible.
◆ ImpliedFeasible indicates the variable or SOS has been presolved out. It will be feasible when all other integer variables or SOS are integer feasible.
◆ Infeasible indicates the variable or SOS is integer infeasible.

See Also

IloCplex.GoalI::IntegerFeasibilityArray,
ControlCallbackI::IntegerFeasibility

Fields

    ImpliedInfeasible
    Feasible
        = CPX_INTEGER_FEASIBLE
    Infeasible
        = CPX_INTEGER_INFEASIBLE
    ImpliedFeasible
        = CPX_IMPLIED_INTEGER_FEASIBLE
Goall::IntegerFeasibilityArray

Category: Inner Type Definition

Definition File: ilocplexi.h

Include Files: ilocplex.h

Synopsis: IloArray< IntegerFeasibility > IntegerFeasibilityArray

Description: This type defines an array type for IloCplex::GoalI::IntegerFeasibility. The fully qualified name of an integer feasibility array is IloCplex::GoalI::IntegerFeasibilityArray.

See Also: IloCplex, GoalI::IntegerFeasibility
**IloCplex::HeuristicCallback**

**Category** Inner Class

**InheritancePath**

**Definition File** ilocplexi.h

**Include Files** ilcplex/ilocplex.h

### Method Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>protected IloCplex::CplexStatus getCplexStatus()</td>
<td></td>
</tr>
<tr>
<td>protected IloAlgorithm::Status getStatus()</td>
<td></td>
</tr>
<tr>
<td>protected IloBool isDualFeasible()</td>
<td></td>
</tr>
<tr>
<td>protected IloBool isPrimalFeasible()</td>
<td></td>
</tr>
<tr>
<td>protected void setBounds(const IloIntVarArray, const IloNumArray, const IloNumArray)</td>
<td></td>
</tr>
<tr>
<td>protected void setBounds(const IloNumVarArray, const IloNumArray, const IloNumArray)</td>
<td></td>
</tr>
<tr>
<td>protected void setBounds(const IloIntVar, IloNum, IloNum)</td>
<td></td>
</tr>
<tr>
<td>protected void setBounds(const IloNumVar, IloNum, IloNum)</td>
<td></td>
</tr>
<tr>
<td>protected void setSolution(const IloIntVarArray, const IloNumArray, IloNum)</td>
<td></td>
</tr>
<tr>
<td>protected void setSolution(const IloIntVarArray, const IloNumArray)</td>
<td></td>
</tr>
</tbody>
</table>
### Inherited methods from `IloCplex::ControlCallbackI`

- `getDownPseudoCost`, `getDownPseudoCost`
- `getFeasibilities`, `getFeasibilities`
- `getFeasibility`, `getFeasibility`
- `getFeasibility`, `getFeasibility`
- `getLB`, `getLB`
- `getLBS`, `getLBS`
- `getNodeData`, `getObjValue`
- `getSlack`, `getSlacks`
- `getUB`, `getUB`
- `getUBs`, `getUBs`
- `getUpPseudoCost`, `getUpPseudoCost`
- `getValue`, `getValue`
- `getValue`, `getValues`
- `getValue`, `getValues`
- `isSOSFeasible`
- `isSOSFeasible`

### Inherited methods from `IloCplex::MIPCallbackI`

- `getBestObjValue`, `getCutoff`
- `getIncumbentObjValue`, `getIncumbentValue`
- `getIncumbentValue`, `getIncumbentValues`
- `getIncumbentValues`, `getMyThreadNum`
- `getNQues`, `getNCovers`
- `getNDisjunctiveCuts`, `getNFlowCovers`
- `getNFlowPaths`, `getNFractionalCuts`
- `getNGUBcovers`, `getNImpliedBounds`
- `getNIterations`, `getNMIRs`
- `getNNodes`, `getNRemainingNodes`
- `getObjCoef`, `getObjCoef`
- `getObjCoefs`, `getObjCoefs`
- `getPriority`, `getPriority`
- `hasIncumbent`
- `hasIncumbent`

<table>
<thead>
<tr>
<th>Method</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>protected void</code></td>
<td><code>setSolution</code> (const IloNumVarArray, const IloNumArray, IloNum)</td>
</tr>
<tr>
<td><code>protected void</code></td>
<td><code>setSolution</code> (const IloNumVarArray, const IloNumArray)</td>
</tr>
<tr>
<td><code>protected IloBool</code></td>
<td><code>solve</code> (IloCplex::Algorithm)</td>
</tr>
</tbody>
</table>
An instance of the class `IloCplex::HeuristicCallbackI` represents a user-written callback in an application that uses an instance of `IloCplex` to solve a mixed integer programming problem (a MIP). When you derive a user-defined class of callbacks, this class offers protected methods for you to:

- give the instance of `IloCplex` a potential new incumbent solution;
- query the instance of `IloCplex` about the solution status for the current node;
- query the instance of `IloCplex` about the variable bounds at the current node;
- change bounds temporarily on a variable or group of variables at the current node;
- re-solve the problem at the node with the changed bounds;
- use all the query functions inherited from parent classes.

In short, this callback allows you to attempt to construct an integer feasible solution at a node and pass it to the invoking instance of `IloCplex` to use as its new incumbent. The API supports you in finding such a solution by allowing you iteratively to change bounds of the variables and re-solve the node relaxation. Changing the bounds in the heuristic callback has no effect on the search beyond the termination of the callback.

The heuristic callback is called after the node relaxation has been solved (including all cuts that may have been generated).

If an attempt is made to access information not available at the node for the invoking instance of `IloCplex`, an exception is thrown.

**See Also**

- `IloCplex`, `IloCplex::Callback`, `IloCplex::CallbackI`, `IloCplex::ControlCallbackI`, `IloCplex::MIPCallbackI`, `ILOHEURISTICCALLBACK0`

**Methods**

```cpp
protected IloCplex::CplexStatus getCplexStatus()
```

This method returns the ILOG CPLEX status of the instance of `IloCplex` at the current node (that is, the state of the optimizer at the node) during the last call to `solve` (which may have been called directly in the callback or by `IloCplex` when processing the node).

The enumeration `IloCplex::CplexStatus` lists the possible status values.
protected IloAlgorithm::Status getStatus()

This method returns the status of the solution found by the instance of IloCplex at the current node during the last call to solve (which may have been called directly in the callback or by IloCplex when processing the node).

The enumeration IloAlgorithm::Status lists the possible status values.

protected IloBool isDualFeasible()

This method returns IloTrue if the solution provided by the last solve call is dual feasible. Note that an IloFalse return value does not necessarily mean that the solution is not dual feasible. It simply means that the relevant algorithm was not able to conclude it was dual feasible when it terminated.

protected IloBool isPrimalFeasible()

This method returns IloTrue if the solution provided by the last solve call is primal feasible. Note that an IloFalse return value does not necessarily mean that the solution is not primal feasible. It simply means that the relevant algorithm was not able to conclude it was primal feasible when it terminated.

protected void setBounds(const IloIntVarArray var, const IloNumArray lb, const IloNumArray ub)

For each variable in the array var, this method sets its upper bound to the corresponding value in the array ub and its lower bound to the corresponding value in the array lb. Setting bounds has no effect beyond the scope of the current invocation of the callback.

protected void setBounds(const IloNumVarArray var, const IloNumArray lb, const IloNumArray ub)

For each variable in the array var, this method sets its upper bound to the corresponding value in the array ub and its lower bound to the corresponding value in the array lb. Setting bounds has no effect beyond the scope of the current invocation of the callback.

protected void setBounds(const IloIntVar var, IloNum lb, IloNum ub)

This method sets the lower bound to lb and the upper bound to ub for the variable var at the current node. Setting bounds has no effect beyond the scope of the current invocation of the callback.

protected void setBounds(const IloNumVar var, IloNum lb, IloNum ub)
This method sets the lower bound to $lb$ and the upper bound to $ub$ for the variable $var$ at the current node. Setting bounds has no effect beyond the scope of the current invocation of the callback.

```java
protected void setSolution(const IloIntVarArray vars, const IloNumArray vals, IloNum obj)
```

For each variable in the array $vars$, this method uses the value in the corresponding element of the array $vals$ to define a heuristic solution to be considered as a new incumbent.

If the user heuristic was successful in finding a new candidate for an incumbent, $setSolution$ can be used to pass it over to $IloCplex$. $IloCplex$ then analyses the solution and, if it is both feasible and better than the current incumbent, uses it as the new incumbent. A solution is specified using arrays $vars$ and $vals$, where $vals[i]$ specifies the solution value for $vars[i]$.

The parameter $obj$ is used to tell $IloCplex$ the objective value of the injected solution. This allows $IloCplex$ to skip the computation of that value, but care must be taken not to provide an incorrect value.

```java
protected void setSolution(const IloIntVarArray vars, const IloNumArray vals)
```

For each variable in the array $vars$, this method uses the value in the corresponding element of the array $vals$ to define a heuristic solution to be considered as a new incumbent.

If the user heuristic was successful in finding a new candidate for an incumbent, $setSolution$ can be used to pass it over to $IloCplex$. $IloCplex$ then analyses the solution and, if it is both feasible and better than the current incumbent, uses it as the new incumbent. A solution is specified using arrays $vars$ and $vals$, where $vals[i]$ specifies the solution value for $vars[i]$.

```java
protected void setSolution(const IloNumVarArray vars, const IloNumArray vals, IloNum obj)
```

For each variable in the array $vars$, this method uses the value in the corresponding element of the array $vals$ to define a heuristic solution to be considered as a new incumbent.

If the user heuristic was successful in finding a new candidate for an incumbent, $setSolution$ can be used to pass it over to $IloCplex$. $IloCplex$ then analyses the solution and, if it is both feasible and better than the current incumbent, uses it as the new incumbent. A solution is specified using arrays $vars$ and $vals$, where $vals[i]$ specifies the solution value for $vars[i]$.
The parameter `obj` is used to tell `IloCplex` the objective value of the injected solution. This allows `IloCplex` to skip the computation of that value, but care must be taken not to provide an incorrect value.

```java
protected void setSolution(const IloNumVarArray vars,
                           const IloNumArray vals)
```

For each variable in the array `vars`, this method uses the value in the corresponding element of the array `vals` to define a heuristic solution to be considered as a new incumbent.

If the user heuristic was successful in finding a new candidate for an incumbent, `setSolution` can be used to pass it over to `IloCplex`. `IloCplex` then analyses the solution and, if it is both feasible and better than the current incumbent, uses it as the new incumbent. A solution is specified using arrays `vars` and `vals`, where `vals[i]` specifies the solution value for `vars[i].`

```java
protected IloBool solve(IloCplex::Algorithm alg)
```

This method can be used to solve the current node relaxation, usually after some bounds have been changed using `setBounds`. By default it uses the dual simplex algorithm, but this can be overwritten using the optional parameter `alg`. See the enumeration `IloCplex::Algorithm` for a list of the available optimizers.
**IloCplex::IISStatus**

**Category**
Inner Enumeration

**Definition File**
ilocplexi.h

**Include Files**
ilcplex/ilocplex.h

**Synopsis**

```cpp
IISStatus{
  Lower,
  Both,
  Upper
};
```

**Description**
The enumeration `IloCplex::IISStatus` lists values that indicate the status that variables and constraints can assume in an irreducibly inconsistent set (IIS) in `IloCplex`.

**See Also**
`IloCplex`, `IloCplex::IISStatusArray`

**Fields**

- `Lower`
  - = CPXIIS_AT_LOWER

- `Both`
  - = CPXIIS_FIXED

- `Upper`
  - = CPXIIS_AT_UPPER
IloCplex::IISStatusArray

Category                     Inner Type Definition
Definition File              ilocplexi.h
Include Files                ilocplex.h
Synopsis                     IloArray< IISStatus > IISStatusArray
Description                  This type defines an array-type for IloCplex::IISStatus. The fully qualified name of a basis status array is IloCplex::IISStatusArray.
See Also                     IloCplex, IloCplex::IISStatus
**IloCplex::IncumbentCallbackI**

**Category** Inner Class

**Inheritance Path**

```
IloCplex::CallbackI
<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>IloCplex::MIPCallbackI</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>IloCplex::IncumbentCallbackI</td>
</tr>
</tbody>
</table>
```

**Definition File** ilocplexi.h

**Include Files** ilcplex/ilocplex.h

### Method Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getNodeData()</td>
<td>protected NodeData *</td>
</tr>
<tr>
<td>getNodeId()</td>
<td>protected NodeId</td>
</tr>
<tr>
<td>getObjValue()</td>
<td>protected IloNum</td>
</tr>
<tr>
<td>getSlack(const IloRange)</td>
<td>protected IloNum</td>
</tr>
<tr>
<td>getSlacks(IloNumArray, const IloRangeArray)</td>
<td>protected void</td>
</tr>
<tr>
<td>getValue(const IloIntVar)</td>
<td>protected IloNum</td>
</tr>
<tr>
<td>getValue(const IloNumVar)</td>
<td>protected IloNum</td>
</tr>
<tr>
<td>getValue(const IloExprArg)</td>
<td>protected IloNum</td>
</tr>
<tr>
<td>getValues(IloNumArray, const IloIntVarArray)</td>
<td>protected void</td>
</tr>
<tr>
<td>getValues(IloNumArray, const IloNumVarArray)</td>
<td>protected void</td>
</tr>
<tr>
<td>reject()</td>
<td>protected void</td>
</tr>
</tbody>
</table>

**Inherited methods from** IloCplex::MIPCallbackI
Description
This callback is called whenever a new potential incumbent is found during branch & cut searches. It allows you to analyze the proposed incumbent and optionally reject it. In this case, CPLEX will continue the branch & cut search. This callback is thus typically combined with a branch callback that instructs CPLEX how to branch on a node after it has found a potential incumbent and thus considered the node solution to be integer feasible.

See Also
IloCplex, IloCplex::Callback, IloCplex::CallbackI, IloCplex::MIPCallbackI

Methods
protected NodeData * getNodeData()
This method retrieves the NodeData object that may have previously been assigned to the current node by the user with method IloCplex::BranchCallbackI::makeBranch. If no data object has been assigned to the current node, 0 will be returned.

protected NodeId getId()
This method returns the NodeId of the current node.

protected IloNum getObjValue()
This method returns the query objective value of the potential incumbent.
IloCplex::IncumbentCallback

protected IloNum getSlack(const IloRange rng)

This method returns the slack value for the range indicated by rng for the potential incumbent.

protected void getSlacks(IloNumArray val,
const IloRangeArray con)

This method puts the slack value for each range in the array of ranges con into the corresponding element of the array val for the potential incumbent. For this CPLEX resizes array val to match the size of array con.

protected IloNum getValue(const IloIntVar var)

This method returns the query value of the variable var in the potential incumbent solution.

protected IloNum getValue(const IloNumVar var)

This method returns the value of the variable var in the potential incumbent solution.

protected IloNum getValue(const IloExprArg expr)

This method returns the value of the expr for the potential incumbent solution.

protected void getValues(IloNumArray val,
const IloIntVarArray vars)

This method returns the query values of the variables in the array vars in the potential incumbent solution and copies them to val. CPLEX automatically resizes the array val to match the size of the array vars.

protected void getValues(IloNumArray val,
const IloNumVarArray vars)

This method returns the query values of the variables in the array vars in the potential incumbent solution and copies them to val. CPLEX automatically resizes the array val to match the length of the array vars.

protected void reject();

This method rejects the proposed incumbent.
**IloCplex::IntParam**

**Category** Inner Enumeration

**Definition File** ilocplexi.h

**Include Files** ilcplex/ilocplex.h

**Synopsis**

```
IloCplex::IntParam
AdvInd,
RootAlg,
NodeAlg,
MIPEmphasis,
AggFill,
AggInd,
BasInterval,
ClockType,
CraInd,
PreDual,
PrePass,
PreCompress,
DPriInd,
PriceLim,
IISInd,
SimDisplay,
ItLim,
NetFind,
PerLim,
PPriInd,
RelInv,
ScaInd,
Threads,
SingLim,
RowGrowth,
ColGrowth,
NzGrowth,
Reduce,
QPNzGrowth,
NzReadLim,
ColReadLim,
RowReadLim,
QPNzReadLim,
SiftDisplay,
SiftAlg,
SiftItLim,
BrDir,
Clique,
CoeRedInd,
Covers,
MIPDisplay,
```
IloCplex::IntParam

MIPInterval,
MIPThreads,
IntSolLim,
NodeFileInd,
NodeLim,
NodeSel,
VarSel,
BndStrenInd,
HeurFreq,
RINSHeur,
SubMIPNodeLim,
MIPordType,
BBInterval,
FlowCovers,
ImplBd,
Probe,
GUBCovers,
StrongCandLim,
StrongItLim,
StrongThreadLim,
FracCand,
FracCuts,
FracPass,
PreslvNd,
FlowPaths,
MIRCuts,
DisjCuts,
AggCutLim,
CutPass,
DiveType,
BarAlg,
BarColNz,
BarDisplay,
BarItLim,
BarMaxCor,
BarOrder,
BarThreads,
BarCrossAlg,
BarStartAlg,
NetItLim,
NetPPriInd,
NetDisplay
}

Description

IloCplex is the class for the CPLEX algorithms in ILOG CPLEX. The enumeration IloCplex::IntParam lists the parameters of CPLEX that require integer values. Use these values with the methods IloCplex::getParam and IloCplex::setParam.
See the reference manual *ILOG CPLEX Parameters* and the *ILOG CPLEX User's Manual* for more information about these parameters. Also see the user's manual for examples of their use.

**See Also**

*IloCplex*

**Fields**

- **AdvInd**
  - `= CPX_PARAM_ADVIND`
- **RootAlg**
  - `= CPX_PARAM_LPMETHOD`
- **NodeAlg**
  - `= CPX_PARAM_SUBALG`
- **MIPEmphasis**
  - `= CPX_PARAM_MIPEMPHASIS`
- **AggFill**
  - `= CPX_PARAM_AGGFILL`
- **AggInd**
  - `= CPX_PARAM_AGGIND`
- **BasInterval**
  - `= CPX_PARAM_BASINTERVAL`
- **ClockType**
  - `= CPX_PARAM_CLOCKTYPE`
- **CraInd**
  - `= CPX_PARAM_CRAIND`
- **PreDual**
  - `= CPX_PARAM_PREDUAL`
- **PrePass**
  - `= CPX_PARAM_PREPASS`
- **PreCompress**
  - `= CPX_PARAM_PRECOMPRESS`
- **DPriInd**
  - `= CPX_PARAM_DPRIIND`
- **PricelIm**
  - `= CPX_PARAM_PRICELIM`
IISInd  
  = CPX_PARAM_IISIND
SimDisplay  
  = CPX_PARAM_SIMDISPLAY
ItLim  
  = CPX_PARAM_ITLIM
NetFind  
  = CPX_PARAM_NETFIND
PerLim  
  = CPX_PARAM_PERLIM
PPriInd  
  = CPX_PARAM_PPRIIND
ReInv  
  = CPX_PARAM_REINV
ScaInd  
  = CPX_PARAM_SCAIND
Threads  
  = CPX_PARAM_THREADS
SingLim  
  = CPX_PARAM_SINGLIM
RowGrowth  
  = CPX_PARAM_ROWGROWTH
ColGrowth  
  = CPX_PARAM_COLGROWTH
NzGrowth  
  = CPX_PARAM_NZGROWTH
Reduce  
  = CPX_PARAM_REDUCE
QPNzGrowth  
  = CPX_PARAM_QPNZGROWTH
NzReadLim  
  = CPX_PARAM_NZREADLIM
ColReadLim
= CPX_PARAM_COLREADLIM
RowReadLim
= CPX_PARAM_ROWREADLIM
QPNzReadLim
= CPX_PARAM_QPNZREADLIM
SiftDisplay
= CPX_PARAM_SIFTDISPLAY
SiftAlg
= CPX_PARAM_SIFTALG
SiftItLim
= CPX_PARAM_SIFTITLIM
BrDir
= CPX_PARAM_BRDIR
Cliques
= CPX_PARAM_CLIQUES
CoeRedInd
= CPX_PARAM_COEREDIND
Covers
= CPX_PARAM_COVERS
MIPDisplay
= CPX_PARAM_MIPDISPLAY
MIPInterval
= CPX_PARAM_MIPINTERVAL
MIPThreads
= CPX_PARAM_MIPTHREADS
IntSollim
= CPX_PARAM_INTSOLLIM
NodeFileInd
= CPX_PARAM_NODEFILEIND
NodeLim
= CPX_PARAM_NODELIM
NodeSel
= CPX_PARAM_NODESEL
VarSel
  = CPX_PARAM_VARSEL

BndStrenInd
  = CPX_PARAM_BNDSTRENIND

HeurFreq
  = CPX_PARAM_HEURFREQ

RINSHeur
  = CPX_PARAM_RINSHEUR

SubMIPNodeLim
  = CPX_PARAM_SUBMIPNODELIM

MIPOrdType
  = CPX_PARAM_MIPORDTYPE

BBInterval
  = CPX_PARAM_BBINTERVAL

FlowCovers
  = CPX_PARAM_FLOWCOVERS

ImplBd
  = CPX_PARAM_IMPLBD

Probe
  = CPX_PARAM_PROBE

GUBCovers
  = CPX_PARAM_GUBCOVERS

StrongCandLim
  = CPX_PARAM_STRONGCANDLIM

StrongItLim
  = CPX_PARAM_STRONGITLIM

StrongThreadLim
  = CPX_PARAM_STRONGTHREADLIM

FracCand
  = CPX_PARAM_FRACCAND

FracCuts
  = CPX_PARAM_FRACCUTS

FracPass
= CPX_PARAM_FRACPASS
PreslvNd
= CPX_PARAM_PRESLVND
FlowPaths
= CPX_PARAM_FLOWPATHS
MIRCuts
= CPX_PARAM_MIRCUTS
DisjCuts
= CPX_PARAM_DISJCUTS
AggCutLim
= CPX_PARAM_AGGCUTLIM
CutPass
= CPX_PARAM_CUTPASS
DiveType
= CPX_PARAM_DIVETYPE
BarAlg
= CPX_PARAM_BARALG
BarColNz
= CPX_PARAM_BARCOLNZ
BarDisplay
= CPX_PARAM_BARDISPLAY
BarItLim
= CPX_PARAM_BARITLIM
BarMaxCor
= CPX_PARAM_BARMAXCOR
BarOrder
= CPX_PARAM_BARORDER
BarThreads
= CPX_PARAM_BARTHREADS
BarCrossAlg
= CPX_PARAM_BARCROSSALG
BarStartAlg
= CPX_PARAM_BARSTARTALG
NetItLim
  = CPX_PARAM_NETITLIM
NetPPriInd
  = CPX_PARAM_NETPPRIIND
NetDisplay
  = CPX_PARAM_NETDISPLAY
**IloCplex::InvalidCutException**

**Category** Inner Class

**InheritancePath**

```
IloCplex::Exception
    IloCplex::InvalidCutException
```

**Definition File**  ilocplexi.h

<table>
<thead>
<tr>
<th><strong>Method Summary</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>public IloConstraint getCut()</td>
</tr>
</tbody>
</table>

**Inherited methods from IloCplex::Exception**

getStatus

**Description**

An instance of this exception is thrown by IloCplex when an attempt is made to add a malformed cut. An example of a malformed cut is one that uses variables that have not been extracted or a cut that is defined with an expression that is not linear.

**Methods**

public IloConstraint getCut()

   Returns the invalid cut that triggered the invoking exception.
IloCplex::LazyConstraintCallbackI

**Category**  
Inner Class

**InheritancePath**

**Definition File**  
ilocplexi.h

**Include Files**  
ilcplex/ilocplex.h

---

### Inherited methods from IloCplex::CutCallbackI

- add
- addLocal

---

### Inherited methods from IloCplex::ControlCallbackI

- ControlCallbackI::getDownPseudoCost
- ControlCallbackI::getFeasibilities
- ControlCallbackI::getFeasibility
- ControlCallbackI::getLB
- ControlCallbackI::getLBS
- ControlCallbackI::getNodeData
- ControlCallbackI::getObjValue
- ControlCallbackI::getSlack
- ControlCallbackI::getSlacks
- ControlCallbackI::getUB
- ControlCallbackI::getUBs
- ControlCallbackI::getUpPseudoCost
- ControlCallbackI::getValue
- ControlCallbackI::isSOSFeasible

---

### Inherited methods from IloCplex::MIPCallbackI


Description

**Note:** This is an advanced routine. Advanced routines typically demand a profound understanding of the algorithms used by ILOG CPLEX. Thus they incur a higher risk of incorrect behavior in your application, behavior that can be difficult to debug. Therefore, ILOG encourages you to consider carefully whether you can accomplish the same task by means of other Callable Library routines instead.

An instance of the class `IloCplex::LazyConstraintCallbackI` represents a user-written callback in an application that uses an instance of `IloCplex` to solve a MIP while generating lazy constraints. `IloCplex` calls the user-written callback after solving each node LP exactly like `IloCplex::CutCallbackI`. In fact, this callback is exactly equivalent to `IloCplex::CutCallbackI` but offers a name more consistently pointing out the difference between lazy constraints and user cuts.
**IloCplex::MIPCallbackI**

**Category** Inner Class

**Inheritance Path**

```
IloCplex::MIPCallbackI
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IloCplex::CallbackI</td>
</tr>
<tr>
<td></td>
<td>IloCplex::ControlCallbackI</td>
</tr>
<tr>
<td></td>
<td>IloCplex::DisjunctiveCutCallbackI</td>
</tr>
<tr>
<td></td>
<td>IloCplex::FlowMIRCutCallbackI</td>
</tr>
<tr>
<td></td>
<td>IloCplex::FractionalCutsCallbackI</td>
</tr>
<tr>
<td></td>
<td>IloCplex::IncumbentCallbackI</td>
</tr>
<tr>
<td></td>
<td>IloCplex::NodeCallbackI</td>
</tr>
<tr>
<td></td>
<td>IloCplex::PruningCallbackI</td>
</tr>
</tbody>
</table>
```

**Definition File** ilocplexi.h

**Include Files** ilcplex/ilocplex.h

**Constructor Summary**

```
protected MIPCallbackI()
```

**Method Summary**

```
protected IloNum MIPCallbackI::getBestObjValue()
protected IloNum MIPCallbackI::getCutoff()
protected IloCplex::BranchDirection MIPCallbackI::getDirection(const IloIntVar)  
protected IloCplex::BranchDirection MIPCallbackI::getDirection(const IloNumVar)
```
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>protected IloNum</td>
<td>getIncumbentObjValue()</td>
</tr>
<tr>
<td>protected IloNum</td>
<td>getIncumbentValue(const IloIntVar)</td>
</tr>
<tr>
<td>protected IloNum</td>
<td>getIncumbentValue(const IloNumVar)</td>
</tr>
<tr>
<td>protected void</td>
<td>getIncumbentValues(IloNumArray, const IloIntVarArray)</td>
</tr>
<tr>
<td>protected void</td>
<td>getIncumbentValues(IloNumArray, const IloNumVarArray)</td>
</tr>
<tr>
<td>protected IloInt</td>
<td>getMyThreadNum()</td>
</tr>
<tr>
<td>protected IloInt</td>
<td>getNcliques()</td>
</tr>
<tr>
<td>protected IloInt</td>
<td>getNcovers()</td>
</tr>
<tr>
<td>protected IloInt</td>
<td>getNdisjunctiveCuts()</td>
</tr>
<tr>
<td>protected IloInt</td>
<td>getNflowCovers()</td>
</tr>
<tr>
<td>protected IloInt</td>
<td>getNflowPaths()</td>
</tr>
<tr>
<td>protected IloInt</td>
<td>getNfractionalCuts()</td>
</tr>
<tr>
<td>protected IloInt</td>
<td>getNGUBcovers()</td>
</tr>
<tr>
<td>protected IloInt</td>
<td>getNimpliedBounds()</td>
</tr>
<tr>
<td>protected IloInt</td>
<td>getNiterations()</td>
</tr>
<tr>
<td>protected IloInt</td>
<td>getNMIRs()</td>
</tr>
<tr>
<td>protected IloInt</td>
<td>getNnodes()</td>
</tr>
<tr>
<td>protected IloInt</td>
<td>getNremainingNodes()</td>
</tr>
<tr>
<td>protected IloNum</td>
<td>getObjCoef(const IloIntVar)</td>
</tr>
<tr>
<td>protected IloNum</td>
<td>getObjCoef(const IloNumVar)</td>
</tr>
<tr>
<td>protected void</td>
<td>getObjCoefs(IloNumArray, const IloIntVarArray)</td>
</tr>
<tr>
<td>protected void</td>
<td>getObjCoefs(IloNumArray, const IloNumVarArray)</td>
</tr>
<tr>
<td>protected IloNum</td>
<td>getPriority(const IloIntVar)</td>
</tr>
<tr>
<td>protected IloNum</td>
<td>getPriority(const IloNumVar)</td>
</tr>
<tr>
<td>protected IloInt</td>
<td>getUserThreads()</td>
</tr>
<tr>
<td>protected IloBool</td>
<td>hasIncumbent()</td>
</tr>
</tbody>
</table>

Inherited methods from **IloCplex::CallbackI**

- CallbackI::abort
- CallbackI::duplicateCallback
- CallbackI::getEnv
- CallbackI::getModel
- CallbackI::getNcols
- CallbackI::getNQCs
- CallbackI::getNrows
**Description**

An instance of the class `IloCplex::MIPCallbackI` represents a user-written callback in an application that uses an instance of `IloCplex` to solve a mixed integer program (MIP). `IloCplex` calls the user-written callback prior to solving each node in branch & cut search. This class offers member functions for accessing an incumbent solution and its objective value from a user-written callback. It also offers methods for accessing priority orders and statistical information such as the number of cuts. Methods are also available to query the number of generated cuts for each type of cut CPLEX generates. See the *ILOG CPLEX User's Manual* for more information about cuts.

The methods of this class are protected for use in deriving a user-written callback class and in implementing the main method there.

If an attempt is made to access information not available to an instance of this class, an exception is thrown. For example, if there is no incumbent, the methods that query about incumbents will throw an exception.

This class also provides the common application programming interface (API) for these callback classes:

- `IloCplex::NodeCallbackI`
- `IloCplex::IncumbentCallbackI`
- `IloCplex::DisjunctiveCutCallbackI`
- `IloCplex::FractionalCutCallbackI`
- `IloCplex::ProbingCallbackI`
- `IloCplex::CutCallbackI`
- `IloCplex::BranchCallbackI`
- `IloCplex::HeuristicCallbackI`
- `IloCplex::SolveCallbackI`

**See Also**

`IloCplex`, `IloCplex::BranchCallbackI`, `IloCplex::Call`, `IloCplex::CallbackI`, `IloCplex::DisjunctiveCutCallbackI`, `IloCplex::FractionalCutCallbackI`, `IloCplex::HeuristicCallbackI`, `IloCplex::IncumbentCallbackI`
IloCplex::MIPCallback

Constructors

protected MIPCallback()

This constructor creates a callback for use in an application that uses an instance of IloCplex to solve a mixed integer program (MIP).

Methods

protected IloNum getBestObjValue()

This method returns the currently best known bound on the optimal solution value of the problem at the time the invoking callback is called by an instance of IloCplex while solving a MIP. When a model has been solved to optimality, this value matches the optimal solution value. Otherwise, this value is computed for a minimization (maximization) problem as the minimum (maximum) objective function value of all remaining unexplored nodes.

protected IloNum getCutoff()

Returns the current cutoff value.

An instance of IloCplex uses the cutoff value (the value of the objective function of the subproblem at a node in the search tree) to decide when to prune nodes from the search tree (that is, when to cut off that node and discard the nodes beyond it). The cutoff value is updated whenever a new incumbent is found.

protected IloCplex::BranchDirection getDirection(const IloIntVar var)

Returns the IloCplex::BranchDirection used for variable var in the current optimization.

protected IloCplex::BranchDirection getDirection(const IloNumVar var)

Returns the IloCplex::BranchDirection used for variable var in the current optimization.

protected IloNum getIncumbentObjValue()

Returns the value of the objective function of the incumbent solution (that is, the best integer solution found so far) at the time the invoking callback is called by an instance of IloCplex while solving a MIP. If there is no incumbent, this method throws an exception.

protected IloNum getIncumbentValue(const IloIntVar var)

Returns the solution value of var in the incumbent solution at the time the invoking callback is called by an instance of IloCplex while solving a MIP. If there is no incumbent, this method throws an exception.

protected IloNum getIncumbentValue(const IloNumVar var)
Returns the solution value of var in the incumbent solution at the time the invoking callback is called by an instance of IloCplex while solving a MIP. If there is no incumbent, this method throws an exception.

protected void getIncumbentValues(IloNumArray val, const IloIntVarArray vars)

Returns the value of each variable in the array vars with respect to the current incumbent solution, and it puts those values into the corresponding array vals. If there is no incumbent, this method throws an exception.

protected void getIncumbentValues(IloNumArray val, const IloNumVarArray vars)

Returns the value of each variable in the array vars with respect to the current incumbent solution, and it puts those values into the corresponding array vals. If there is no incumbent, this method throws an exception.

protected IloInt getMyThreadNum()

Returns the identifier of the parallel thread being currently executed. This number is between 0 (zero) and the value returned by the method getUserThreads()-1.

protected IloInt getNcliques()

Returns the total number of clique cuts that have been added to the model so far during the current optimization.

protected IloInt getNcovers()

Returns the total number of cover cuts that have been added to the model so far during the current optimization.

protected IloInt getNdisjunctiveCuts()

Returns the total number of disjunctive cuts that have been added to the model so far during the current optimization.

protected IloInt getNflowCovers()

Returns the total number of flow cover cuts that have been added to the model so far during the current optimization.

protected IloInt getNflowPaths()

Returns the total number of flow path cuts that have been added to the model so far during the current optimization.

protected IloInt getNfractionalCuts()

Returns the total number of fractional cuts that have been added to the model so far during the current optimization.

protected IloInt getNGUBcovers()
Returns the total number of GUB cover cuts that have been added to the model so far during the current optimization.

protected IloInt getNimpliedBounds()

Returns the total number of implied bound cuts that have been added to the model so far during the current optimization.

protected IloInt getNiterations()

Returns the total number of iterations executed so far during the current optimization to solve the node relaxations.

protected IloInt getNMIRs()

Returns the total number of MIR cuts that have been added to the model so far during the current optimization.

protected IloInt getNnodes()

Returns the number of nodes already processed in the current optimization.

protected IloInt getNremainingNodes()

Returns the number of nodes left to explore in the current optimization.

protected IloNum getObjCoef(const IloIntVar var)

Returns the linear objective coefficient for var in the model currently being solved.

protected IloNum getObjCoef(const IloNumVar var)

Returns the linear objective coefficient for var in the model currently being solved.

protected void getObjCoefs(IloNumArray val, const IloIntVarArray vars)

Puts the linear objective coefficient of each of the variables in the array vars into the corresponding element of the array vals.

protected void getObjCoefs(IloNumArray val, const IloNumVarArray vars)

Puts the linear objective coefficient of each of the variables in the array vars into the corresponding element of the array vals.

protected IloNum getPriority(const IloIntVar sos)

Returns the branch priority used for variable var in the current optimization.

protected IloNum getPriority(const IloNumVar sos)

Returns the branch priority used for variable var in the current optimization.

protected IloInt getUserThreads()

Returns the total number of parallel threads currently running.

protected IloBool hasIncumbent()
Returns IloTrue if an integer feasible solution has been found, or, equivalently, if an incumbent solution is available at the time the invoking callback is called by an instance of IloCplex while solving a MIP.
MIPCallbackI::NodeData

Category Inner Class

InheritancePath

Definition File ilocplexi.h

Include Files ilocplex.h

Method Summary

| Public virtual IloAny | getDataType() |

Description Objects of (a subclass of) this class can be attached to created nodes in a branch callback with one of the IloCplex::BranchCallbackI::makeBranch methods. This allows the user to associate arbitrary data with the nodes. The destructor must be implemented to delete all such data. It will typically be called by IloCplex when a node is discarded, either because it has been processed or because it is pruned.

See Also IloCplex::MIPCallbackI, ILOBRANCHCALLBACK0

Methods

public virtual IloAny getDataType() 

IloCplex does not use this method. It is provided as a convenience for the user to help manage different types of NodeData subclasses.
IloCplex::MIPEmphasisType

Category: Inner Enumeration

Definition File: ilocplexi.h

Include Files: ilcplex/ilocplex.h

Synopsis:

```cpp
MIPEmphasisType {
    MIPEmphasisBalanced,
    MIPEmphasisOptimality,
    MIPEmphasisFeasibility,
    MIPEmphasisBestBound,
    MIPEmphasisHiddenFeas
};
```

Description:

The enumeration `IloCplex::MIPEmphasisType` lists the values that the MIP emphasis parameter `IloCplex::MIPEmphasis` can assume in an instance of `IloCplex` for use when it is solving MIP problems. Use these values with the method `IloCplex::setParam(IloCplex::MIPEmphasis, value)` when you set MIP emphasis.

With the default setting of `IloCplex::MIPEmphasisBalance`, IloCplex tries to compute the branch & cut algorithm in such a way as to find a proven optimal solution quickly. For a discussion about various settings, refer to the *ILOG CPLEX User’s Manual*.

See the reference manual *ILOG CPLEX Parameters* and the *ILOG CPLEX User’s Manual* for more information about these parameters. Also see the user's manual for examples of their use.

See Also: `IloCplex`

Fields:

- `MIPEmphasisBalanced` = CPX_MIPEMPHASIS_BALANCED
- `MIPEmphasisOptimality` = CPX_MIPEMPHASIS_OPTIMALITY
- `MIPEmphasisFeasibility` = CPX_MIPEMPHASIS_FEASIBILITY
- `MIPEmphasisBestBound` = CPX_MIPEMPHASIS_BESTBOUND
- `MIPEmphasisHiddenFeas`
= CPX_MIPEMPHASIS_HIDDENFEAS
IloCplex::MultipleConversionException

Category
Inner Class

InheritancePath

Definition File
ilocplexi.h

Method Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>public IloConversion</td>
<td>getConversion()</td>
</tr>
<tr>
<td>public const IloNumVarArray</td>
<td>getVariables()</td>
</tr>
</tbody>
</table>

Inherited methods from **IloCplex::Exception**

getStatus

Description
An instance of this exception is thrown by IloCplex when there is an attempt to convert the type of a variable with more than one IloConversion object at a time, while it is being extracted by IloCplex.

Methods

public IloConversion getConversion()

This method returns the offending IloConversion object.

public const IloNumVarArray getVariables()

This method returns an array of variables to which too many type conversions have been applied.
**IloCplex::MultipleObjException**

**Category** Inner Class

**InheritancePath**

```
IloCplex::Exception
IloCplex::MultipleObjException
```

**Definition File** ilocplexi.h

**Inherited methods from** IloCplex::Exception
- getStatus

**Description**
An instance of this exception is thrown by IloCplex when there is an attempt to use more than one objective function in a model extracted by IloCplex.
**IloCplex::NetworkCallbackI**

**Category**
Inner Class

**Inheritance Path**
- IloCplex::CallbackI
- IloCplex::NetworkCallbackI

**Definition File**
ilocplexi.h

**Include Files**
ilcplex/ilcplex.h

### Constructor Summary

<table>
<thead>
<tr>
<th>Access</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>protected</td>
<td>NetworkCallbackI()</td>
<td></td>
</tr>
</tbody>
</table>

### Method Summary

<table>
<thead>
<tr>
<th>Access</th>
<th>Type</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>protected</td>
<td>IloNum</td>
<td>getInfeasibility()</td>
<td></td>
</tr>
<tr>
<td>protected</td>
<td>IloInt</td>
<td>getNiterations()</td>
<td></td>
</tr>
<tr>
<td>protected</td>
<td>IloNum</td>
<td>getObjValue()</td>
<td></td>
</tr>
<tr>
<td>protected</td>
<td>IloBool</td>
<td>isFeasible()</td>
<td></td>
</tr>
</tbody>
</table>

**Inherited methods from IloCplex::CallbackI**

- CallbackI::abort
- CallbackI::duplicateCallback
- CallbackI::getEnv
- CallbackI::getModel
- CallbackI::getNcols
- CallbackI::getNQCs
- CallbackI::getNrows
- CallbackI::main

**Description**

An instance of the class IloCplex::NetCallbackI represents a user-written callback in an application that uses an instance of IloCplex with the network optimizer. The callback is executed each time the network optimizer issues a log file message.
The methods of this class are protected for use in deriving a user-written callback class and in implementing the main method there.

If an attempt is made to access information not available to an instance of this class, an exception is thrown.

See Also
IloCplex, IloCplex::Callback, IloCplex::CallbackI, ILONETWORKCALLBACK

Constructors
protected NetworkCallbackI()
This constructor creates a callback for use with the network optimizer.

Methods
protected IloNum getInfeasibility()
This method returns the current primal infeasibility measure of the network solution in the instance of IloCplex at the time the invoking callback is executed.

protected IloInt getNiterations()
This method returns the number of network simplex iterations completed so far by an instance of IloCplex at the invoking callback is executed.

protected IloNum getObjValue()
This method returns the current objective value of the network solution in the instance of IloCplex at the time the invoking callback is executed.

protected IloBool isFeasible()
This method returns IloTrue if the current network solution is primal feasible.
IloCplex::NodeCallbackI

Category: Inner Class

Inheritance Path:

- IloCplex::CallbackI
- IloCplex::MIPCallbackI
- IloCplex::NodeCallbackI

Definition File: ilocplexi.h
Include Files: ilcplex/ilocplex.h

Constructor Summary:

| protected NodeCallbackI() |

Method Summary:

| protected IloNumVar getBranchVar(NodeId) | protected IloNumVar getBranchVar(int) |
| protected IloInt getDepth(NodeId)       | protected IloInt getDepth(int)       |
| protected IloNum getEstimatedObjValue(NodeId) | protected IloNum getEstimatedObjValue(int) |
| protected IloNum getInfeasibilitySum(NodeId) | protected IloNum getInfeasibilitySum(int) |
| protected IloInt getNinfeasibilities(NodeId) | protected IloInt getNinfeasibilities(int) |
| protected NodeData * getNodeData(NodeId) | protected NodeData * getNodeData(int) |
| protected NodeId getNodeId(NodeId)      | protected NodeId getNodeId(int)      |
| protected IloNum getObjValue(NodeId)    | protected IloNum getObjValue(int)    |
Description

An instance of the class IloCplex::NodeCallbackI represents a user-written callback in an application that uses an instance of IloCplex to solve a mixed integer programming problem (a MIP). The methods of this class enable you (from a user-derived callback class) to query the instance of IloCplex about the next node that it plans to select in the branch & cut search, and optionally to override this selection by specifying a different node.

When an instance of this callback executes, the invoking instance of IloCplex still has n = getNremainingNodes (inherited from IloCplex::MIPCallbackI) nodes left to process. These remaining nodes are numbered from 0 (zero) to (n - 1). For that reason, the same node may have a different number each time an instance of NodeCallbackI is called. To identify a node uniquely, an instance of IloCplex also assigns a unique NodeId to each node. That unique identifier remains unchanged throughout the search. The method getNodeId(int i) allows you to access the

Inherited methods from IloCplex::MIPCallbackI

- getBestObjValue
- getCutoff
- getDirection
- getIncumbentObjValue
- getIncumbentValue
- getIncumbentValues
- getMyThreadNum
- getNcliques
- getNcovers
- getNdisjunctiveCuts
- getNflowCovers
- getNflowPaths
- getNfractionalCuts
- getNGUBcovers
- getNimpliedBounds
- getNiterations
- getNMIRs
- getNnodes
- getNremainingNodes
- getObjCoef
- getObjCoefs
- getMyThreadNum
- getPriority
- getPriority
- getUserThreads
- hasIncumbent

Inherited methods from IloCplex::CallbackI

- abort
- duplicateCallback
- getEnv
- getModel
- getNcols
- getNQCs
- getNrows
- main
NodeId for each of the remaining nodes (0 to n-1). Such a query allows you to associate data with individual nodes.

The methods of this class are protected for use in deriving a user-written callback class and in implementing the main method there.

If an attempt is made to access information not available to an instance of this class, an exception is thrown.

See Also
IloCplex, IloCplex::Callback, IloCplex::CallbackI, IloCplex::MIPCallbackI, ILONODECALLBACK0

Constructors
protected NodeCallbackI()

This constructor creates a callback for use in an application with user-defined node selection inquiry during branch & cut searches.

Methods
protected IloNumVar getBranchVar(NodeId nodeid)

This method returns the variable that was branched on last when CPLEX created the node with the identifier nodeid. If that node has been created by branching on a constraint or on multiple variables, 0 (zero) will be returned.

protected IloNumVar getBranchVar(int node)

Returns the variable that was branched on last when creating the node indicated by the index number node. If that node has been created by branching on a constraint or on multiple variables, 0 (zero) will be returned.

protected IloInt getDepth(NodeId nodeid)

This method returns the depth of the node in the search tree for the node with the identifier nodeid. The root node has depth 0 (zero). The depth of other nodes is defined recursively as the depth of their parent node plus one. In other words, the depth of a node is its distance in terms of the number of branches from the root.

protected IloInt getDepth(int node)

This method returns the depth of the node in the search tree. The root node has depth 0 (zero). The depth of other nodes is defined recursively as the depth of their parent node plus one. In other words, the depth of a node is its distance in terms of the number of branches from the root.

protected IloNum getEstimatedObjValue(NodeId nodeid)

This method returns the estimated objective value of the node with the identifier nodeid.

protected IloNum getEstimatedObjValue(int node)

This method returns the estimated objective value of the node indicated by the index number node.
protected IloNum getInfeasibilitySum(NodeId nodeid)
This method returns the sum of infeasibility measures at the node with the identifier nodeid.

protected IloNum getInfeasibilitySum(int node)
This method returns the sum of infeasibility measures at the node indicated by the index number node.

protected IloInt getNinfeasibilities(NodeId nodeid)
This method returns the number of infeasibilities at the node with the identifier nodeid.

protected IloInt getNinfeasibilities(int node)
This method returns the number of infeasibilities at the node indicated by the index number node.

protected NodeData * getNodeData(NodeId nodeid)
This method retrieves the NodeData object that may have previously been assigned by the user to the node with the identifier nodeid with one of the methods IloCplex::BranchCallbackI::makeBranch. If no data object has been assigned to the that node, 0 (zero) will be returned.

protected NodeData * getNodeData(int node)
This method retrieves the NodeData object that may have previously been assigned to the node with index node by the user with the method IloCplex::BranchCallbackI::makeBranch. If no data object has been assigned to the specified node, 0 (zero) will be returned.

protected NodeId getNodeId(int node)
This method returns the node identifier of the node indicated by the index number node. During branch & cut, an instance of IloCplex assigns node identifiers sequentially from 0 (zero) to (getNodes - 1) as it creates nodes. Within a search, these node identifiers are unique throughout the duration of that search. However, at any point, the remaining nodes, (that is, the nodes that have not yet been processed) are stored in an array in an arbitrary order. This method returns the identifier of the node stored at position node in that array.

protected IloInt getNodeNumber(NodeId nodeid)
Returns the current index number of the node indicated by the node identifier nodeid.

protected IloNum getObjValue(NodeId nodeid)
This method returns the objective value of the node with the identifier nodeid.

protected IloNum getObjValue(int node)
This method returns the objective value of the node indicated by the index number node.

protected void selectNode(NodeId nodeid)

This method selects the node with the identifier nodeid and sets it as the next node to process in the branch & cut search. The invoking instance of IloCplex uses the specified node as the next node to process.

protected void selectNode(int node)

This method selects the node indicated by its index number node and sets it as the next node to process in the branch & cut search. The parameter node must be an integer between 0 (zero) and (getNremainingNodes - 1).

The invoking instance of IloCplex uses the specified node as the next node to process.
**IloCplex::NodeEvaluator**

**Category**  
Inner Class

**Inheritance Path**

**Definition File**  
ilocplexi.h

**Include Files**  
ilcplex/ilcplex.h

**Constructor Summary**

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>NodeEvaluator()</td>
</tr>
<tr>
<td>public</td>
<td>NodeEvaluator(IloCplex::NodeEvaluatorI *)</td>
</tr>
<tr>
<td>public</td>
<td>NodeEvaluator(const NodeEvaluator &amp;)</td>
</tr>
</tbody>
</table>

**Method Summary**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>getImpl()</td>
</tr>
<tr>
<td>public IloCplex::NodeEvaluatorI *</td>
<td>operator=(const NodeEvaluator &amp;)</td>
</tr>
</tbody>
</table>

**Description**

Node evaluators can be used to control the node selection strategy during goal-controlled search, that is, the order in which nodes are processed during branch & cut search using IloCplex goals. This is the handle class for objects of type IloCplex::NodeEvaluatorI, which allow you to control the node selection scheme.

IloCplex::NodeEvaluatorI objects are reference-counted. This means that every instance of IloCplex::NodeEvaluatorI keeps track of how many handle objects refer to it. When this number drops to 0, the IloCplex::NodeEvaluatorI object is automatically deleted. As a consequence, whenever you deal with node evaluators, you must maintain a handle object rather than just a pointer to the implementation object. Otherwise, you risk ending up with a pointer to an implementation object that has already been deleted.
See Also

IloCplex, IloCplex::NodeEvaluatorI

Constructors

public NodeEvaluator()

The empty constructor creates a new evaluator containing no pointers to an implementation object.

dependent NodeEvaluator(IloCplex::NodeEvaluatorI * impl)

This constructor creates a new evaluator with a pointer to an implementation. It increases the reference count of impl by one.

dependent NodeEvaluator(const NodeEvaluator & eval)

This copy constructor increments the reference count of the implementation object referenced by eval by one.

Methods

public IloCplex::NodeEvaluatorI * getImpl()

Queries the implementation object.

dependent NodeEvaluator operator=(const NodeEvaluator & eval)

The assignment operator increases the reference count of the implementation object of eval. If the invoking handle referred to an implementation object before the assignment operation, its reference count is decreased. If this decrement reduces the reference count to 0 (zero), the implementation object is deleted.
IloCplex::NodeEvaluatorI

Category Inner Class

InheritancePath

Definition File ilocplexi.h

Include Files ilcplex/ilocplex.h

Constructor Summary

| public | NodeEvaluatorI() |

Method Summary

| public virtual NodeEvaluatorI * | duplicateEvaluator() |
| public virtual IloNum | evaluate() |
| protected IloNumVar | getBranchVar() |
| protected IloNum | getDepth() |
| protected IloNum | getEstimatedObjValue() |
| protected IloNum | getInfeasibilitySum() |
| protected IloInt | getNinfeasibilities() |
| protected NodeId | getNodeId() |
| protected IloNum | getObjValue() |
| public virtual void | init() |
| public virtual IloBool | subsume(IloNum, IloNum) |

Description IloCplex::NodeEvaluatorI is the base class for implementing node evaluators. Node evaluators allow you to control the node selection strategy for a subtree by assigning values to the nodes. By default, IloCplex selects the node with the lowest value when choosing the next node to process during branch & cut search. This behavior can be altered by overwriting method subsume.
To implement your own node evaluator, you need to create a subclass of IloCplex::NodeEvaluatorI and implement methods evaluate and duplicateEvaluator. The method evaluate must be implemented to compute and return a value for a given node. The protected methods of class IloCplex::NodeEvaluatorI can be called to query information about the node in order to compute this value. Each node is evaluated only once, after which the value is attached to the node until the node is processed or pruned.

The duplicateEvaluator method is called by IloCplex when a copy of the evaluator must be created for use in parallel branch & cut search. Thus the implementation must simply create and return a copy of the evaluator itself—calling the copy constructor will work in most circumstances.

Node evaluators are applied to a search defined by a goal with the method IloCplex::Apply. The node selection strategy will be applied only to the subtree defined by the goal passed to Apply. Using IloCplex::Apply, you can assign different node selection strategies to different subtrees. You can also assign multiple node selection strategies to subtrees. In this case, node selection strategies applied first have precedence over those assigned later.

If no node evaluators are added, IloCplex uses the node selection strategy as controlled by the NodeSel parameter.

See Also
IloCplex, IloCplex::NodeEvaluator

Constructors
public NodeEvaluatorI()

This constructor creates a node selector for use in an application with a user-defined node selection strategy to solve a MIP.

Methods
public virtual NodeEvaluatorI * duplicateEvaluator()

This method must be implemented by the user to return a copy of the invoking object. It is called internally to duplicate the current node evaluator for parallel branch & cut search. This method is not called for a particular node, so the get methods cannot be used.

public virtual IloNum evaluate()

This method must be implemented by the user to return a value for a given node. When this method is called, the node evaluator is initialized to the node for which to compute the value. Information about this node can be obtained by the get methods of IloCplex::NodeEvaluatorI. Returning IloInfinity instructs IloCplex to discard the node being evaluated.

protected IloNumVar getBranchVar()

This method returns the variable that IloCplex branched on when creating the node being evaluated from its parent. If the node has been generated with a more complex
branch, 0 (zero) will be returned instead. This method can be called only from the methods init and evaluate.

protected IloNum getDepth()

This method returns the depth in the search tree of the node currently being evaluated. The root node is depth 0 (zero); the depth of the current node is its distance from the root, or equivalently, the number of branches taken to get from the root node to the current node. This member function can be called only from the methods init and evaluate.

protected IloNum getEstimatedObjValue()

This method returns the estimated objective value for the node being evaluated. It can be called only from the methods init and evaluate.

protected IloNum getInfeasibilitySum()

This method returns the sum of infeasibility measures at the node being evaluated. It can be called only from the methods init and evaluate.

protected IloInt getNinfeasibilities()

This method returns the number of infeasibilities at the node being evaluated. It can be called only from the methods init and evaluate.

protected NodeId getNodeId()

This method returns the node identifier of the node being evaluated. It can be called only from the methods init and evaluate.

protected IloNum getObjValue()

This method returns the objective value of the node being evaluated. It can be called only from the methods init and evaluate.

public virtual void init()

This method is called by IloCplex immediately before the first time evaluate is called for a node, allowing you to initialize the evaluator based on that node. Information about the current node can be queried by calling the get methods of IloCplex::NodeEvaluatorI.

public virtual IloBool subsume(IloNum evalBest, IloNum evalCurrent)

IloCplex maintains a candidate node for selection as the next node to process. When choosing the next node, it compares the candidate to all other nodes. If a given node and the candidate node are governed by the same evaluator, IloCplex calls subsume to determine whether the node should become the new candidate. The arguments passed to the subsume call are:

◆ the value previously assigned by the method evaluate to the candidate node as parameter evalBest, and
the value previously assigned by the method evaluate to the node under investigation as parameter evalCurrent.

By default, this method returns IloTrue if evalCurrent>evalBest. Overwriting this function allows you to change this selection scheme.
**IloCplex::NodeSelect**

**Category**
Inner Enumeration

**Definition File**
ilocplexi.h

**Include Files**
ilcplex/ilocplex.h

**Synopsis**

```cpp
NodeSelect{
  DFS,
  BestBound,
  BestEst,
  BestEstAlt
};
```

**Description**
The enumeration `IloCplex::NodeSelect` lists values that the parameter `IloCplex::NodeSel` can assume in `IloCplex`. Use these values with the method `IloCplex::setParam(IloCplex::NodeSel, value)`.

See the reference manual *ILOG CPLEX Parameters* and the *ILOG CPLEX User's Manual* for more information about these parameters. Also see the user's manual for examples of their use.

**See Also**
IloCplex

**Fields**

```cpp
DFS
  = CPX_NODESEL_DFS

BestBound
  = CPX_NODESEL_BESTBOUND

BestEst
  = CPX_NODESEL_BESTEST

BestEstAlt
  = CPX_NODESEL_BESTEST_ALT
```
IloCplex::NumParam

Category       Inner Enumeration
Definition File ilocplex1.h
Include Files  ilcplex/ilocplex.h

Synopsis

```
NumParam {
    EpMrk,
    EpOpt,
   EpPer,
    EpRHS,
    NetEpOpt,
    NetEqRHS,
    TiLim,
    BtTol,
    CutLo,
    CutUp,
    EpGap,
    EpInt,
    EpAdjap,
    ObjDif,
    ObjLLim,
    ObjULim,
    RelObjDif,
    CutsFactor,
    TreLim,
    WorkMem,
    BarEpComp,
    BarQCPEpComp,
    BarGrowth,
    BarObjRng
};
```

Description

The enumeration IloCplex::NumParam lists the parameters of CPLEX that require numeric values. Use these values with the member functions: `IloCplex::getParam` and `IloCplex::setParam`.

See the reference manual ILOG CPLEX Parameters information about these parameters. Also see the ILOG CPLEX User's Manual for more examples of their use.

See Also

IloCplex

Fields

- **EpMrk**
  - = CPX_PARAM_EPMRK
- **EpOpt**
ILO CPLEX C++ API 9.0 REFERENCE MANUAL

IloCplex::NumParam

= CPX_PARAM_EPOPT
EpPer
= CPX_PARAM_EPPER
EpRHS
= CPX_PARAM_EPRHS
NetEpOpt
= CPX_PARAM_NETEPOPT
NetEpRHS
= CPX_PARAM_NETEPRHS
TiLim
= CPX_PARAM_TILIM
BtTol
= CPX_PARAM_BTTOL
CutLo
= CPX_PARAM_CUTLO
CutUp
= CPX_PARAM_CUTUP
EpGap
= CPX_PARAM_EPGAP
EpInt
= CPX_PARAM_EPINT
EpAGap
= CPX_PARAM_EPAGAP
ObjDif
= CPX_PARAM_OBJDIF
ObjLLim
= CPX_PARAM_OBJLLIM
ObjULim
= CPX_PARAM_OBJULIM
RelObjDif
= CPX_PARAM_RELOBJDIF
CutsFactor
= CPX_PARAM_CUTSFACCTOR
IloCplex::NumParam

TreLim
= CPX_PARAM_TRELIM

WorkMem
= CPX_PARAM_WORKMEM

BarEpComp
= CPX_PARAM_BAREPCOMP

BarQCPEpComp
= CPX_PARAM_BARQCPEPCOMP

BarGrowth
= CPX_PARAM_BARGROWTH

BarObjRng
= CPX_PARAM_BAROBJRNG
IloCplex::PWLFormulation

Category Inner Enumeration

Definition File ilocplexi.h

Include Files ilocplex.h

Synopsis

PWLFormulation{
  LambdaSOS,
  LambdaBin,
  Disjunctive,
  Delta
};

Description

The enumeration PWLFormulation defines symbolic constants that represent various ways of representing piecewise linear expressions in a MIP. Which of these ways to use is controlled by the method setPWLFormulation. All of these formulations have the same theoretical strength. However, in practice, the effectiveness for solving a particular model may vary dramatically, depending on the formulation being used. These are the various formulations:

- **Disjunctive**, the default, represents a tight formulation of the logical disjunction underlying the function that is being made piecewise linear.

- **LambdaSOS** represents a transformation of a piecewise linear expression in which a variable is created for each breakpoint, and a special ordered set of type 2 (SOS2) is added for those variables.

- **LambdaBin** represents a transformation of a piecewise linear expression similar to LambdaSOS in which, however, the SOS2 constraint is replaced by a set of linear constraints defined over additional binary variables.

- **Delta** indicates a cumulative formulation.

Fields

LambdaSOS
LambdaBin
Disjunctive
Delta
**IloCplex::PresolveCallbackI**

**Category**  
Inner Class

**Inheritance Path**

```
IloCplex::CallbackI
```

**Definition File**  
ilocplexi.h

**Include Files**  
ilcplex/ilocplex.h

### Constructor Summary

<table>
<thead>
<tr>
<th>Protected</th>
<th>PresolveCallbackI()</th>
</tr>
</thead>
</table>

### Method Summary

- **getNaggregations()**  
- **getNmodifiedCoeffs()**  
- **getNremovedCols()**  
- **getNremovedRows()**

### Inherited methods from **IloCplex::CallbackI**

- `CallbackI::abort`, `CallbackI::duplicateCallback`, `CallbackI::getEnv`, `CallbackI::getMatrixMode`, `CallbackI::getNcols`, `CallbackI::getNQCs`, `CallbackI::getNrows`, `CallbackI::main`

**Description**

An instance of a class derived from `PresolveCallbackI` represents a user-written callback in an application that uses an instance of `IloCplex`. The callback is called periodically during presolve. This class enables you to access information about the effects of presolve on the model extracted for the instance of `IloCplex`. For example, there are member functions that return the number of rows or columns removed from the model.
model, the number of variables that have been aggregated, and the number of coefficients that have changed as a result of presolve.

The constructor and methods of this class are protected for use in deriving a user-written callback class and in implementing the main method there.

If an attempt is made to access information not available to an instance of this class, an exception is thrown.

See Also IloCplex, IloCplex::Callback, IloCplex::CallbackI, ILOPRESOLVECALLBACK0

Constructors

protected PresolveCallbackI()

This constructor creates a callback for use in presolve.

Methods

protected IloInt getNaggregations()

This method returns the number of aggregations performed by presolve at the time the callback is executed.

protected IloInt getNmodifiedCoeffs()

This method returns the number of coefficients modified by presolve at the time the callback is executed.

protected IloInt getNremovedCols()

This method returns the number of columns removed by presolve at the time the callback is executed.

protected IloInt getNremovedRows()

This method returns the number of rows removed by presolve at the time the callback is executed.
IloCplex::PrimalPricing

Category           Inner Enumeration
Definition File    ilocplexi.h
Include Files      ilcplex/ilocplex.h
Synopsis
PrimalPricing{
PPriIndPartial,
PPriIndAuto,
PPriIndDevex,
PPriIndSteep,
PPriIndSteepQStart,
PPriIndFull
};

Description
The enumeration IloCplex::PrimalPricing lists values that the primal pricing parameter IloCplex::PPriInd can assume in IloCplex for use with the primal simplex algorithm. Use these values with the method IloCplex::setParam(IloCplex::PPriInd, value) when setting the primal pricing indicator.

See the reference manual ILGO CPLEX Parameters and the ILOG CPLEX User's Manual for more information about these parameters. Also see the user's manual for examples of their use.

See Also
IloCplex

Fields
PPriIndPartial
  = CPX_PPRIIND_PARTIAL
PPriIndAuto
  = CPX_PPRIIND_AUTO
PPriIndDevex
  = CPX_PPRIIND_DEVEX
PPriIndSteep
  = CPX_PPRIIND_STEEP
PPriIndSteepQStart
  = CPX_PPRIIND_STEEPQSTART
PPriIndFull
  = CPX_PPRIIND_FULL
**IloCplex::ProbingCallbackI**

**Category** Inner Class

**InheritancePath**

```
<table>
<thead>
<tr>
<th>IloCplex::CallbackI</th>
</tr>
</thead>
<tbody>
<tr>
<td>IloCplex::MIPCallbackI</td>
</tr>
<tr>
<td>IloCplex::ProbingCallbackI</td>
</tr>
</tbody>
</table>
```

**Definition File** ilocplexi.h

**Include Files** ilcplex/ilocplex.h

### Constructor Summary

<table>
<thead>
<tr>
<th>protected</th>
<th>ProbingCallbackI()</th>
</tr>
</thead>
</table>

### Method Summary

<table>
<thead>
<tr>
<th>protected IloInt</th>
<th>getPhase()</th>
</tr>
</thead>
<tbody>
<tr>
<td>protected IloNum</td>
<td>getProgress()</td>
</tr>
</tbody>
</table>

### Inherited methods from IloCplex::MIPCallbackI
IloCplex::ProbingCallbackI

Description
An instance of the class IloCplex::ProbingCallbackI represents a user-written callback in an application that uses an instance of IloCplex to solve a mixed integer programming problem (a MIP). This class offers a method to check on the progress of probing operation.

The methods of this class are protected for use in deriving a user-written callback class and in implementing the main method there.

If an attempt is made to access information not available to an instance of this class, an exception is thrown.

See Also
IloCplex, IloCplex::Callback, IloCplex::CallbackI, IloCplex::MIPCallbackI, ILOPROBINGCALLBACK0

Constructors
protected ProbingCallbackI()
This constructor creates a callback for use in an application when probing.

Methods
protected IloInt getPhase()
This method returns the current phase of probing.
protected IloNum getProgress()
This method returns the fraction of completion of the current probing phase.
IloCplex::Quality

Category: Inner Enumeration

Definition File: ilocplexi.h

Include Files: ilcplex/ilocplex.h

Synopsis:

```
IloCplex::Quality
{
  MaxPrimalInfeas,
  MaxScaledPrimalInfeas,
  SumPrimalInfeas,
  SumScaledPrimalInfeas,
  MaxDualInfeas,
  MaxScaledDualInfeas,
  SumDualInfeas,
  SumScaledDualInfeas,
  MaxIntInfeas,
  SumIntInfeas,
  MaxPrimalResidual,
  MaxScaledPrimalResidual,
  SumPrimalResidual,
  SumScaledPrimalResidual,
  MaxDualResidual,
  MaxScaledDualResidual,
  SumDualResidual,
  SumScaledDualResidual,
  MaxCompSlack,
  SumCompSlack,
  MaxX,
  MaxScaledX,
  MaxPi,
  MaxScaledPi,
  MaxSlack,
  MaxScaledSlack,
  MaxRedCost,
  MaxScaledRedCost,
  SumX,
  SumScaledX,
  SumPi,
  SumScaledPi,
  SumSlack,
  SumScaledSlack,
  SumRedCost,
  SumScaledRedCost,
  Kappa,
  Objgap,
  ObjGap,
  DualObj,
  DualObj
}```
The enumeration `IloCplex::Quality` lists types of quality measures that can be queried for a solution with method `IloCplex::getQuality`.

See the group optim.cplex.solutionquality in the *ILOG CPLEX Callable Library Reference Manual* for more information about these values. Also see the *ILOG CPLEX User's Manual* for examples of their use.

### See Also
- `IloCplex`

### Fields
```
MaxPrimalInfeas = CPX_MAX_PRIMAL_INFEAS
MaxScaledPrimalInfeas = CPX_MAX_SCALED_PRIMAL_INFEAS
SumPrimalInfeas = CPX_SUM_PRIMAL_INFEAS
SumScaledPrimalInfeas = CPX_SUM_SCALED_PRIMAL_INFEAS
MaxDualInfeas = CPX_MAX_DUAL_INFEAS
MaxScaledDualInfeas = CPX_MAX_SCALED_DUAL_INFEAS
SumDualInfeas = CPX_SUM_DUAL_INFEAS
SumScaledDualInfeas = CPX_SUM_SCALED_DUAL_INFEAS
MaxIntInfeas = CPX_MAX_INT_INFEAS
SumIntInfeas = CPX_SUM_INT_INFEAS
MaxPrimalResidual = CPX_MAX_PRIMAL_RESIDUAL
MaxScaledPrimalResidual = CPX_MAX_SCALED_PRIMAL_RESIDUAL
```


\[
\begin{align*}
\text{SumPrimalResidual} &= \text{CPX\_SUM\_PRIMAL\_RESIDUAL} \\
\text{SumScaledPrimalResidual} &= \text{CPX\_SUM\_SCALED\_PRIMAL\_RESIDUAL} \\
\text{MaxDualResidual} &= \text{CPX\_MAX\_DUAL\_RESIDUAL} \\
\text{MaxScaledDualResidual} &= \text{CPX\_MAX\_SCALED\_DUAL\_RESIDUAL} \\
\text{SumDualResidual} &= \text{CPX\_SUM\_DUAL\_RESIDUAL} \\
\text{SumScaledDualResidual} &= \text{CPX\_SUM\_SCALED\_DUAL\_RESIDUAL} \\
\text{MaxCompSlack} &= \text{CPX\_MAX\_COMP\_SLACK} \\
\text{SumCompSlack} &= \text{CPX\_SUM\_COMP\_SLACK} \\
\text{MaxX} &= \text{CPX\_MAX\_X} \\
\text{MaxScaledX} &= \text{CPX\_MAX\_SCALED\_X} \\
\text{MaxPi} &= \text{CPX\_MAX\_PI} \\
\text{MaxScaledPi} &= \text{CPX\_MAX\_SCALED\_PI} \\
\text{MaxSlack} &= \text{CPX\_MAX\_SLACK} \\
\text{MaxScaledSlack} &= \text{CPX\_MAX\_SCALED\_SLACK} \\
\text{MaxRedCost} &= \text{CPX\_MAX\_RED\_COST} \\
\text{MaxScaledRedCost} &= \text{CPX\_MAX\_SCALED\_RED\_COST} \\
\text{SumX} &= \text{CPX\_SUM\_X}
\end{align*}
\]
= CPX_SUM_X
SumScaledX
= CPX_SUM_SCALED_X
SumPi
= CPX_SUM_PI
SumScaledPi
= CPX_SUM_SCALED_PI
SumSlack
= CPX_SUM_SLACK
SumScaledSlack
= CPX_SUM_SCALED_SLACK
SumRedCost
= CPX_SUM_RED_COST
SumScaledRedCost
= CPX_SUM_SCALED_RED_COST
Kappa
= CPX_KAPPA
Objgap
DEPRECATED use ObjGap instead
ObjGap
= CPX_OBJ_GAP
Dualobj
DEPRECATED use DualObj instead
DualObj
= CPX_DUAL_OBJ
Primalobj
DEPRECATED use PrimalObj instead
PrimalObj
= CPX_PRIMAL_OBJ
IloCplex::SearchLimit

Category: Inner Class

Inheritance Path:

**Definition File** ilocplexi.h

**Include Files** ilocplex/ilocplex.h

### Constructor Summary

<table>
<thead>
<tr>
<th>Public Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SearchLimit()</td>
<td>Constructor for SearchLimit()</td>
</tr>
<tr>
<td>SearchLimit(IloCplex::SearchLimitI *)</td>
<td>Constructor for SearchLimit with pointer</td>
</tr>
<tr>
<td>SearchLimit(const SearchLimit &amp;)</td>
<td>Constructor for SearchLimit with copy</td>
</tr>
</tbody>
</table>

### Method Summary

<table>
<thead>
<tr>
<th>Public Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IloCplex::SearchLimitI * getImpl()</td>
<td>Get implementation of SearchLimit</td>
</tr>
<tr>
<td>operator=(const SearchLimit &amp;)</td>
<td>Copy assignment for SearchLimit</td>
</tr>
</tbody>
</table>

### Description

Search limits can be used to impose limits on the exploration of certain subtrees during branch & cut search. Search limits are implemented in the class IloCplex::SearchLimitI. This is the handle class for CPLEX search limits.

The search limit objects are reference-counted. This means an instance of IloCplex::SearchLimitI keeps track of how many handle objects refer to it. If this number drops to 0, the IloCplex::SearchLimitI object is automatically deleted. As a consequence, whenever you deal with a search limit, you must maintain a handle object rather than only a pointer to the implementation object. Otherwise, you risk ending up with a pointer to an implementation object that has already been deleted.

### See Also

IloCplex, IloCplex::SearchLimitI

### Constructors

public SearchLimit()
The default constructor creates a new search limit with 0 implementation object pointer.

public SearchLimit(IloCplex::SearchLimitI * impl)

This constructor creates a new search limit with a pointer to an implementation. It increases the reference count of impl by one.

public SearchLimit(const SearchLimit & limit)

This copy constructor increments the reference count of the implementation object referenced by limit by one.

Methods

public IloCplex::SearchLimitI * getImpl()

Queries the implementation object of the invoking search limit.

public SearchLimit operator=(const SearchLimit & limit)

The assignment operator increases the reference count of the implementation object of limit. If the invoking handle referred to an implementation object before the assignment operation, its reference count is decreased. If this reduces the reference count to 0, the implementation object is deleted.
**IloCplex::SearchLimitI**

**Category** Inner Class

**InheritancePath**

**Definition File** ilocplexI.h

**Include Files** ilcplex/ilocplex.h

### Constructor Summary

<table>
<thead>
<tr>
<th>public</th>
<th>SearchLimitI()</th>
</tr>
</thead>
</table>

### Method Summary

<table>
<thead>
<tr>
<th>public virtual IloBool</th>
<th>check()</th>
</tr>
</thead>
<tbody>
<tr>
<td>public virtual SearchLimitI*</td>
<td>duplicateLimit()</td>
</tr>
<tr>
<td>public virtual void</td>
<td>init()</td>
</tr>
</tbody>
</table>

**Description**

IloCplex::SearchLimitI is the base class for implementing user-defined search limits. To do so, you must subclass IloCplex::SearchLimitI and implement methods check and duplicateLimit. You may optionally implement method init. The method check must return IloTrue when the limit is reached and IloFalse otherwise. The method duplicateLimit must return a copy of the invoking object to be used in parallel search.

Whenever method check is called by IloCplex, the search limit object is first initialized to a node, referred to as the current node. Information about the current node can be queried by calling the get methods of class IloCplex::SearchLimitI.

Search limits are applied to subtrees defined by goals with method IloCplex::LimitSearch. For example:

```cpp
IloGoal limitGoal = IloCplex::LimitSearch(cplex, goal1, limit);
```
creates a goal `limitGoal` which branches as specified by `goal1` until the limit specified by `limit` is reached. Only the nodes created by `goal1` (or any of the goals created by it later) are subjected to the search limit. For example, if you created two branches with the goal

```cpp
OrGoal(limitGoal, goal2);
```

only the subtree defined by `goal1` is subject to the search limit `limit`; the subtree defined by `goal2` is not.

The ability to specify search limits for subtrees means that it is possible for certain branches to be subject to more than one search limit. Nodes with multiple search limits attached to them are processed only if none of the search limits has been reached, or, in other words, if all the search limits return `IloFalse` when method `check` is called by `IloCplex`.

Each time CPLEX uses a search limit, it is duplicated first. If you use the same instance of your limit in different branches, it will be duplicated first, the copy will be passed to the corresponding node, and `init` method will be called on the copy.

**See Also**

- `IloCplex`, `IloCplex::SearchLimit`

**Constructors**

```cpp
public SearchLimitI()
```

The default constructor creates a new instance of `SearchLimitI`.

**Methods**

```cpp
public virtual IloBool check()
```

This method is called for every node subjected to the invoking search limit before evaluating the node. If it returns `IloTrue`, the node is pruned, or, equivalently, the search below that node is discontinued. Thus, users implementing search limits must implement this method to return `IloTrue` if the search limit has been reached and `IloFalse` otherwise.

```cpp
public virtual SearchLimitI * duplicateLimit()
```

This method is called internally to duplicate the current search limit. Users must implement it in a subclass to return a copy of the invoking object.

```cpp
public virtual void init()
```

This method is called by `IloCplex` right before the first time `check` is called for a node and allows you to initialize the limit based on that node.
IloCplex::SimplexCallbackI

Category Inner Class

InheritancePath

Definition File ilocplexi.h

Include Files ilcplex/ilocplex.h

<table>
<thead>
<tr>
<th>Constructor Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>protected SimplexCallbackI()</td>
</tr>
</tbody>
</table>

Inherited methods from IloCplex::ContinuousCallbackI

- ContinuousCallbackI::getDualInfeasibility
- ContinuousCallbackI::getInfeasibility
- ContinuousCallbackI::getNiterations
- ContinuousCallbackI::getObjValue
- ContinuousCallbackI::isDualFeasible
- ContinuousCallbackI::isFeasible

Inherited methods from IloCplex::CallbackI

- CallbackI::abort
- CallbackI::duplicateCallback
- CallbackI::getEnv
- CallbackI::getModel
- CallbackI::getNcols
- CallbackI::getNQCs
- CallbackI::getNrows
- CallbackI::main

Description

An instance of the class IloCplex::SimplexCallbackI represents a user-written callback in an application that uses an instance of IloCplex to solve a problem by means of the simplex optimizer. For more information on the simplex optimizer, see the
ILOG CPLEX User's Manual. IloCplex calls the user-written callback after each iteration during optimization with the simplex algorithm.

The constructor of this class is protected for use in deriving a user-written callback class and in implementing the main method there.

If an attempt is made to access information not available to an instance of this class, an exception is thrown.

**See Also**

IloCplex, IloCplex::Callback, IloCplex::CallbackI, IloCplex::ContinuousCallbackI, ILOSIMPLEXCALLBACK0

**Constructors**

protected SimplexCallbackI()

This constructor creates a callback for use in an application of the simplex optimizer.
IloCplex::SolveCallbackI

Category            Inner Class

InheritancePath

Definition File    ilocplexi.h

Include Files      ilcplex/ilocplex.h

Constructor Summary

| protected | SolveCallbackI() |

Method Summary

| protected | IloCplex::CplexStatus | getCplexStatus() |
| protected | IloAlgorithm::Status  | getStatus()      |
| protected | IloBool               | isDualFeasible() |
| protected | IloBool               | isPrimalFeasible() |
| protected void | setVectors(const IloNumArray, const IloIntVarArray, const IloNumArray, const IloRangeArray) | |
| protected void | setVectors(const IloNumArray, const IloNumVarArray, const IloNumArray, const IloRangeArray) | |
| protected | IloBool               | solve(IloCplex::Algorithm) |
| protected | IloBool               | useSolution()   |
Inherited methods from **IloCplex::ControlCallbackI**

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>ControlCallbackI::getDownPseudoCost</td>
</tr>
<tr>
<td>ControlCallbackI::getFeasibilities</td>
</tr>
<tr>
<td>ControlCallbackI::getFeasibility</td>
</tr>
<tr>
<td>ControlCallbackI::getLB</td>
</tr>
<tr>
<td>ControlCallbackI::getLBs</td>
</tr>
<tr>
<td>ControlCallbackI::getNodeData</td>
</tr>
<tr>
<td>ControlCallbackI::getObjValue</td>
</tr>
<tr>
<td>ControlCallbackI::getSlack</td>
</tr>
<tr>
<td>ControlCallbackI::getSlacks</td>
</tr>
<tr>
<td>ControlCallbackI::getUB</td>
</tr>
<tr>
<td>ControlCallbackI::getUBs</td>
</tr>
<tr>
<td>ControlCallbackI::getUpPseudoCost</td>
</tr>
<tr>
<td>ControlCallbackI::getValue</td>
</tr>
<tr>
<td>ControlCallbackI::getValue</td>
</tr>
<tr>
<td>ControlCallbackI::isSOSFeasible</td>
</tr>
</tbody>
</table>

Inherited methods from **IloCplex::MIPCallbackI**

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIPCallbackI::getBestObjValue</td>
</tr>
<tr>
<td>MIPCallbackI::getCutoff</td>
</tr>
<tr>
<td>MIPCallbackI::getDirection</td>
</tr>
<tr>
<td>MIPCallbackI::getDirection</td>
</tr>
<tr>
<td>MIPCallbackI::getIncumbentObjectiveValue</td>
</tr>
<tr>
<td>MIPCallbackI::getIncumbentValue</td>
</tr>
<tr>
<td>MIPCallbackI::getIncumbentValues</td>
</tr>
<tr>
<td>MIPCallbackI::getIncumbentValues</td>
</tr>
<tr>
<td>MIPCallbackI::getMyThreadNum</td>
</tr>
<tr>
<td>MIPCallbackI::getNcliques</td>
</tr>
<tr>
<td>MIPCallbackI::getNcovers</td>
</tr>
<tr>
<td>MIPCallbackI::getNdisjunctiveCuts</td>
</tr>
<tr>
<td>MIPCallbackI::getNflowCovers</td>
</tr>
<tr>
<td>MIPCallbackI::getNflowPaths</td>
</tr>
<tr>
<td>MIPCallbackI::getNfractionalCuts</td>
</tr>
<tr>
<td>MIPCallbackI::getNGUBcovers</td>
</tr>
<tr>
<td>MIPCallbackI::getNimpliedBounds</td>
</tr>
<tr>
<td>MIPCallbackI::getNiterations</td>
</tr>
<tr>
<td>MIPCallbackI::getNMIRs</td>
</tr>
<tr>
<td>MIPCallbackI::getNnodes</td>
</tr>
<tr>
<td>MIPCallbackI::getNremainingNodes</td>
</tr>
<tr>
<td>MIPCallbackI::getObjCoef</td>
</tr>
<tr>
<td>MIPCallbackI::getObjCoef</td>
</tr>
<tr>
<td>MIPCallbackI::getObjCoefs</td>
</tr>
<tr>
<td>MIPCallbackI::getObjCoefs</td>
</tr>
<tr>
<td>MIPCallbackI::getPriority</td>
</tr>
<tr>
<td>MIPCallbackI::getUserThreads</td>
</tr>
<tr>
<td>MIPCallbackI::hasIncumbent</td>
</tr>
</tbody>
</table>

Inherited methods from **IloCplex::CallbackI**

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>CallbackI::abort</td>
</tr>
<tr>
<td>CallbackI::duplicateCallback</td>
</tr>
<tr>
<td>CallbackI::getEnv</td>
</tr>
<tr>
<td>CallbackI::getModel</td>
</tr>
<tr>
<td>CallbackI::getNcols</td>
</tr>
<tr>
<td>CallbackI::getNQCs</td>
</tr>
<tr>
<td>CallbackI::getNrows</td>
</tr>
<tr>
<td>CallbackI::main</td>
</tr>
</tbody>
</table>
Description

An instance of the class IloCplex::SolveCallbackI can be used to solve node relaxations during branch & cut search. It allows setting a starting point for the solve or selecting the algorithm on a per-node basis.

The methods of this class are protected for use in deriving a user-written callback class and in implementing the main method there.

If an attempt is made to access information not available to an instance of this class, an exception is thrown.

See Also

IloCplex, IloCplex::Callback, IloCplex::CallbackI, IloCplex::ControlCallbackI, ILOSOLVECALLBACK0

Constructors

protected SolveCallbackI()

This constructor creates a callback for use in an application for solving the node LPs during branch & cut searches.

Methods

protected IloCplex::CplexStatus getCplexStatus()

This method returns the ILOG CPLEX status of the instance of IloCplex at the current node (that is, the state of the optimizer at the node) during the last call to solve (which may have been called directly in the callback or by IloCplex when processing the node).

The enumeration IloCplex::CplexStatus lists the possible status values.

protected IloAlgorithm::Status getStatus()

This method returns the status of the solution found by the instance of IloCplex at the current node during the last call to solve (which may have been called directly in the callback or by IloCplex when processing the node).

The enumeration IloAlgorithm::Status lists the possible status values.

protected IloBool isDualFeasible()

This method returns IloTrue if the solution provided by the last solve call is dual feasible. Note that an IloFalse return value does not necessarily mean that the solution is not dual feasible. It simply means that the relevant algorithm was not able to conclude it was dual feasible when it terminated.

protected IloBool isPrimalFeasible()

This method returns IloTrue if the solution provided by the last solve call is primal feasible. Note that an IloFalse return value does not necessarily mean that the solution is not primal feasible. It simply means that the relevant algorithm was not able to conclude it was primal feasible when it terminated.

protected void setVectors(const IloNumArray x,
                          const IloIntVarArray var,
```
const IloNumArray pi,
const IloRangeArray rng)
```

This method allows a user to specify a starting point for the following invocation of the `solve` method in a solve callback. Zero can be passed for any of the parameters. However, if \( x \) is not zero, then \( var \) must not be zero either. Similarly, if \( pi \) is not zero, then \( rng \) must not be zero either.

For all variables in \( var.x[i] \) specifies the starting value for the variable \( var[i] \). Similarly, for all ranged constraints specified in \( rng.pi[i] \) specifies the starting dual value for \( rng[i] \).

This information is exploited at the next call to `solve`, to construct a starting point for the algorithm.

```
protected void setVectors(const IloNumArray x,
const IloNumVarArray var,
const IloNumArray pi,
const IloRangeArray rng)
```

This method allows a user to specify a starting point for the following invocation of the `solve` method in a solve callback. Zero can be passed for any of the parameters. However, if \( x \) is not zero, then \( var \) must not be zero either. Similarly, if \( pi \) is not zero, then \( rng \) must not be zero either.

For all variables in \( var.x[i] \) specifies the starting value for the variable \( var[i] \). Similarly, for all ranged constraints specified in \( rng.pi[i] \) specifies the starting dual value for \( rng[i] \).

This information is exploited at the next call to `solve`, to construct a starting point for the algorithm.

```
protected IloBool solve(IloCplex::Algorithm alg)
```

This method uses the algorithm \( alg \) to solve the current node LP. See `IloCplex::Algorithm` for a choice of algorithms to use.

```
protected void useSolution()
```

A call to this method instructs `IloCplex` to use the solution generated with this callback.

If `useSolution` is not called, `IloCplex` uses the algorithm selected with the parameters `IloCplex::RootAlg` for the solution of the root, or `IloCplex::NodeAlg` to solve the node.
IloCplex::StringParam

Category Inner Enumeration

Definition File ilocplexi.h

Include Files ilcplex/ilocplex.h

Synopsis

StringParam {
    WorkDir
};

Description

The enumeration IloCplex::StringParam lists the parameters of CPLEX that require a character string as a value. Use these values with the methods IloCplex::getParam and IloCplex::setParam.

See the reference manual ILOG CPLEX Parameter Table and the ILOG CPLEX User’s Manual for more information about these parameters. Also see the user’s manual for examples of their use.

See Also IloCplex

Fields

WorkDir

= CPX_PARAM_WORKDIR
IloCplex::UnknownExtractableException

Category Inner Class

InheritancePath

Definition File ilocplexi.h

Method Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public IloExtractable getExtractable()</td>
<td>This method returns the offending extractable object.</td>
</tr>
</tbody>
</table>

Inherited methods from **IloCplex::Exception**

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>getStatus</td>
</tr>
</tbody>
</table>

Description An instance of this exception is thrown by IloCplex when an operation is attempted using an extractable that has not been extracted.

Methods

public IloExtractable getExtractable()

This method returns the offending extractable object.
## IloCplex::UserCutCallbackI

**Category**  
Inner Class

**InheritancePath**

**Definition File**  
ilocplexi.h

**Include Files**  
ilplex/ilocplex.h

### Inherited methods from IloCplex::CutCallbackI

- `add`, `addLocal`

### Inherited methods from IloCplex::ControlCallbackI

- `ControlCallbackI::getDownPseudoCost`, `ControlCallbackI::getDownPseudoCost`,  
- `ControlCallbackI::getFeasibilities`, `ControlCallbackI::getFeasibilities`,  
- `ControlCallbackI::getFeasibility`, `ControlCallbackI::getFeasibility`,  
- `ControlCallbackI::getFeasibility`, `ControlCallbackI::getFeasibility`,  
- `ControlCallbackI::getLB`, `ControlCallbackI::getLB`,  
- `ControlCallbackI::getLBS`, `ControlCallbackI::getLBS`,  
- `ControlCallbackI::getNodeData`, `ControlCallbackI::getObjValue`,  
- `ControlCallbackI::getSlack`, `ControlCallbackI::getSlacks`,  
- `ControlCallbackI::getUB`, `ControlCallbackI::getUB`,  
- `ControlCallbackI::getUBs`, `ControlCallbackI::getUBs`,  
- `ControlCallbackI::getUpPseudoCost`, `ControlCallbackI::getUpPseudoCost`,  
- `ControlCallbackI::getValue`, `ControlCallbackI::getValue`,  
- `ControlCallbackI::getValue`, `ControlCallbackI::getValue`,  
- `ControlCallbackI::isSOSFeasible`, `ControlCallbackI::isSOSFeasible`

### Inherited methods from IloCplex::MIPCallbackI


Description

**Note:** This is an advanced routine. Advanced routines typically demand a profound understanding of the algorithms used by ILOG CPLEX. Thus they incur a higher risk of incorrect behavior in your application, behavior that can be difficult to debug. Therefore, ILOG encourages you to consider carefully whether you can accomplish the same task by means of other Callable Library routines instead.

An instance of the class `IloCplex::UserCutCallbackI` represents a user-written callback in an application that uses an instance of `IloCplex` to solve a MIP while generating user cuts to tighten the LP relaxation. `IloCplex` calls the user-written callback after solving each node LP exactly like `IloCplex::CutCallbackI`. It differs from `IloCplex::CutCallbackI` only in that constraints added in a `UserCutCallbackI` must be real cuts in the sense that omitting them does not affect the feasible region of the model under consideration.
IloCplex::VariableSelect

Category: Inner Enumeration

Definition File: ilocplexi.h

Include Files: ilcplex/ilocplex.h

Synopsis:
```
VariableSelect {
  MinInfeas,
  DefaultVarSel,
  MaxInfeas,
  Pseudo,
  Strong,
  PseudoReduced
};
```

Description:
The enumeration IloCplex::VariableSelect lists values that the parameter IloCplex::VarSel can assume in IloCplex. Use these values with the method IloCplex::setParam(IloCplex::VarSel, value).

See the reference manual ILOG CPLEX Parameters and the ILOG CPLEX User's Manual for more information about these parameters. Also see the user's manual for examples of their use.

See Also:
IloCplex

Fields:
- MinInfeas
  = CPX_VARSEL_MININFEAS
- DefaultVarSel
  = CPX_VARSEL_DEFAULT
- MaxInfeas
  = CPX_VARSEL_MAXINFEAS
- Pseudo
  = CPX_VARSEL_PSEUDO
- Strong
  = CPX_VARSEL_STRONG
- PseudoReduced
  = CPX_VARSEL_PSEUDOREDUCED
The API of ILOG CPLEX Piecewise Linear Formulations for users of C++.

### Enumerations Summary

<table>
<thead>
<tr>
<th>Enumeration</th>
</tr>
</thead>
<tbody>
<tr>
<td>IloCplex::PWLFormulation</td>
</tr>
</tbody>
</table>

### Description

#### Piecewise Linearity in ILOG CPLEX

Some problems are most naturally represented by constraints over functions that are not purely linear but consist of linear segments. Such functions are sometimes known as piecewise linear.

#### How to Define a Piecewise Linear Function

To define a piecewise linear function in Concert Technology, you need these components:

- the variable of the piecewise linear function;
- the breakpoints of the piecewise linear function;
- the slope of each segment (that is, the rate of increase or decrease of the function between two breakpoints);
- the geometric coordinates of at least one point of the function.

In other words, for a piecewise linear function of n breakpoints, you need to know n+1 slopes. Typically, the breakpoints of a piecewise linear function are specified as an array of numeric values. The slopes of its segments are indicated as an array of numeric values as well. The geometric coordinates of at least one point of the function must also be specified. Then in Concert Technology, those details are brought together in an instance of the class IloPiecewiseLinear.

Another way to specify a piecewise linear function is to give the slope of the first segment, two arrays for the coordinates of the breakpoints, and the slope of the last segment.

For examples of these ways of defining a piecewise linear function, see this topic in the ILOG CPLEX User’s Manual.

#### Discontinuous Piecewise Linear Functions

Intuitively, in a continuous piecewise linear function, the endpoint of one segment has the same coordinates as the initial point of the next segment. There are piecewise linear functions, however, where two consecutive breakpoints may have the same x
coordinate but differ in the value of $f(x)$. Such a difference is known as a step in the piecewise linear function, and such a function is known as discontinuous.

Syntactically, a step is represented in this way:

- The value of the first point of a step in the array of slopes is the height of the step.
- The value of the second point of the step in the array of slopes is the slope of the function after the step.

By convention, a breakpoint belongs to the segment that starts at that breakpoint.

In Concert Technology, a discontinuous piecewise linear function is represented as an instance of the class IloPiecewiseLinear (the same class as used for continuous piecewise linear functions).

For examples of discontinuous piecewise linear functions, see this topic in the ILOG CPLEX User's Manual.

Using IloPiecewiseLinear

Whether it represents a continuous or a discontinuous piecewise linear function, an instance of IloPiecewiseLinear behaves like a floating-point expression. That is, you may use it in a term of a linear expression or in a constraint added to a model (an instance of IloModel).
IloCplex::PWLFormulation

Category
Inner Enumeration

Definition File
ilocplexi.h

Include Files
ilocplex.h

Synopsis
PWLFormulation {
  LambdaSOS,
  LambdaBin,
  Disjunctive,
  Delta
};

Description
The enumeration PWLFormulation defines symbolic constants that represent various ways of representing piecewise linear expressions in a MIP. Which of these ways to use is controlled by the method setPWLFormulation. All of these formulations have the same theoretical strength. However, in practice, the effectiveness for solving a particular model may vary dramatically, depending on the formulation being used. These are the various formulations:

- **Disjunctive**, the default, represents a tight formulation of the logical disjunction underlying the function that is being made piecewise linear.

- **LambdaSOS** represents a transformation of a piecewise linear expression in which a variable is created for each breakpoint, and a special ordered set of type 2 (SOS2) is added for those variables.

- **LambdaBin** represents a transformation of a piecewise linear expression similar to LambdaSOS in which, however, the SOS2 constraint is replaced by a set of linear constraints defined over additional binary variables.

- **Delta** indicates a cumulative formulation.

Fields
LambdaSOS
LambdaBin
Disjunctive
Delta

Group optim.cplex.cpp.logical

The API of ILOG CPLEX Logical Formulations for users of C++.

Description
What Are Logical Constraints?
For ILOG CPLEX, a logical constraint combines linear constraints by means of logical operators, such as logical and, logical or, negation (that is, not), conditional statements (that is, if ... then ...) to express complex relations between linear constraints. ILOG CPLEX can also handle certain logical expressions appearing within a linear constraint. One such logical expression is the minimum of a set of variables. Another such logical expression is the absolute value of a variable.

In C++ applications, the class IloCplex can extract modeling objects to solve a wide variety of MIPs and LPs. Under some conditions, a problem expressed in terms of logical constraints may be equivalent to a continuous LP, rather than a MIP. In such a case, there is no need for branching during the search for a solution. Whether a problem (or parts of a problem) represented by logical terms can be modeled and solved by LP depends on the shape of those logical terms. In this context, shape means convex or concave in the formal, mathematical sense.

For more about convexity, see that topic in the ILOG CPLEX User's Manual.

In fact, the class IloCplex can extract logical constraints as well as some logical expressions. The logical constraints that IloCplex can extract are these:

- IloAnd which can also be represented by the overloaded operator & &;
- IloOr which can also be represented by the overloaded operator | |
- negation as represented by the overloaded operator !;
- IloIfThen
- == (that is, the equivalence relation)
- != (that is, the exclusive-or relation)

For examples of logical constraints in ILOG CPLEX, see the ILOG CPLEX User's Manual.

### Group optim.cplex.advanced.cpp

The advanced methods of the API of ILOG CPLEX for users of C++.

### Classes Summary

<table>
<thead>
<tr>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>IloCplex::LazyConstraintCallbackI</td>
</tr>
<tr>
<td>IloCplex::UserCutCallbackI</td>
</tr>
</tbody>
</table>

ILOG CPLEX C++ API 9.0 Reference Manual 231
Description

These are advanced methods. Advanced methods typically demand a profound understanding of the algorithms used by ILOG CPLEX. Thus they incur a higher risk of incorrect behavior in your application, behavior that can be difficult to debug. Therefore, ILOG encourages you to consider carefully whether you can accomplish the same task by means of other methods instead.
**IloCplex::LazyConstraintCallbackI**

**Category**  
Inner Class

**Inheritance Path**

```
 IloCplex::CallbackI
    ↓
IloCplex::MIPCallbackI
    ↓
IloCplex::ControlCallbackI
    ↓
IloCplex::CutCallbackI
    ↓
IloCplex::LazyConstraintCallbackI
```

**Definition File**  
ilocplexi.h

**Include Files**  
ilcplex/ilocplex.h

---

**Inherited methods from IloCplex::CutCallbackI**

- add
- addLocal

---

**Inherited methods from IloCplex::ControlCallbackI**

- ControlCallbackI::getDownPseudoCost
- ControlCallbackI::getDownPseudoCost
- ControlCallbackI::getFeasibilities
- ControlCallbackI::getFeasibilities
- ControlCallbackI::getFeasibility
- ControlCallbackI::getFeasibility
- ControlCallbackI::getFeasibility
- ControlCallbackI::getLB
- ControlCallbackI::getLB
- ControlCallbackI::getLBS
- ControlCallbackI::getLBS
- ControlCallbackI::getNodeData
- ControlCallbackI::getObjValue
- ControlCallbackI::getSlack
- ControlCallbackI::getSlacks
- ControlCallbackI::getUB
- ControlCallbackI::getUB
- ControlCallbackI::getUBs
- ControlCallbackI::getUBs
- ControlCallbackI::getValue
- ControlCallbackI::getValue
- ControlCallbackI::getValues
- ControlCallbackI::isSOSFeasible
- ControlCallbackI::isSOSFeasible
An instance of the class IloCplex::LazyConstraintCallbackI represents a user-written callback in an application that uses an instance of IloCplex to solve a MIP while generating lazy constraints. IloCplex calls the user-written callback after solving each node LP exactly like IloCplex::CutCallbackI. In fact, this callback is exactly equivalent to IloCplex::CutCallbackI but offers a name more consistently pointing out the difference between lazy constraints and user cuts.

Inherited methods from IloCplex::MIPCallbackI

- MIPCallbackI::getBestObjValue
- MIPCallbackI::getCutoff
- MIPCallbackI::getDirection
- MIPCallbackI::getIncumbentObjValue
- MIPCallbackI::getIncumbentValue
- MIPCallbackI::getIncumbentValues
- MIPCallbackI::getMyThreadNum
- MIPCallbackI::getNcliques
- MIPCallbackI::getNcovers
- MIPCallbackI::getNdissjunctiveCuts
- MIPCallbackI::getNflowCovers
- MIPCallbackI::getNflowPaths
- MIPCallbackI::getNfractionalCuts
- MIPCallbackI::getNGUBcovers
- MIPCallbackI::getNimpliedBounds
- MIPCallbackI::getNiterations
- MIPCallbackI::getNremainingNodes
- MIPCallbackI::getObjCoef
- MIPCallbackI::getObjCoefs
- MIPCallbackI::getPriority
- MIPCallbackI::getUserThreads
- MIPCallbackI::hasIncumbent

Inherited methods from IloCplex::CallbackI

- CallbackI::abort
- CallbackI::duplicateCallback
- CallbackI::getEnv
- CallbackI::getModel
- CallbackI::getNcols
- CallbackI::getNQCs
- CallbackI::getNrows
- CallbackI::main

Note: This is an advanced routine. Advanced routines typically demand a profound understanding of the algorithms used by ILOG CPLEX. Thus they incur a higher risk of incorrect behavior in your application, behavior that can be difficult to debug. Therefore, ILOG encourages you to consider carefully whether you can accomplish the same task by means of other Callable Library routines instead.
**IloCplex::UserCutCallbackI**

**Category**  
Inner Class

**Inheritance Path**

- IloCplex::CallbackI
- IloCplex::MIPCallbackI
- IloCplex::ControlCallbackI
- IloCplex::UserCutCallbackI

**Definition File**  
ilocplexi.h

**Include Files**  
ilcplex/ilocplex.h

### Inherited methods from **IloCplex::CutCallbackI**

- add
- addLocal

### Inherited methods from **IloCplex::ControlCallbackI**

- ControlCallbackI::getDownPseudoCost
- ControlCallbackI::getDownPseudoCost
- ControlCallbackI::getFeasibilities
- ControlCallbackI::getFeasibilities
- ControlCallbackI::getFeasibility
- ControlCallbackI::getFeasibility
- ControlCallbackI::getLB
- ControlCallbackI::getLB
- ControlCallbackI::getLBs
- ControlCallbackI::getLBs
- ControlCallbackI::getNodeData
- ControlCallbackI::getObjValue
- ControlCallbackI::getSlack
- ControlCallbackI::getSlacks
- ControlCallbackI::getUB
- ControlCallbackI::getUB
- ControlCallbackI::getUBs
- ControlCallbackI::getUBs
- ControlCallbackI::getUpPseudoCost
- ControlCallbackI::getUpPseudoCost
- ControlCallbackI::getValue
- ControlCallbackI::getValue
- ControlCallbackI::getValue
- ControlCallbackI::isSOSFeasible
IloCplex::UserCutCallbackI

Description

An instance of the class IloCplex::UserCutCallbackI represents a user-written callback in an application that uses an instance of IloCplex to solve a MIP while generating user cuts to tighten the LP relaxation. IloCplex calls the user-written callback after solving each node LP exactly like IloCplex::CutCallbackI. It differs from IloCplex::CutCallbackI only in that constraints added in a UserCutCallbackI must be real cuts in the sense that omitting them does not affect the feasible region of the model under consideration.

Inherited methods from IloCplex::MIPCallbackI

MIPCallbackI::getBestObjValue, MIPCallbackI::getCutoff, MIPCallbackI::getDirection, MIPCallbackI::getIncumbentObjValue, MIPCallbackI::getIncumbentValue, MIPCallbackI::getIncumbentValues, MIPCallbackI::getMyThreadNum, MIPCallbackI::getNcliques, MIPCallbackI::getNcovers, MIPCallbackI::getNdisjunctiveCuts, MIPCallbackI::getNflowCovers, MIPCallbackI::getNflowPaths, MIPCallbackI::getNfractionalCuts, MIPCallbackI::getNGUBcovers, MIPCallbackI::getNimpliedBounds, MIPCallbackI::getNiterations, MIPCallbackI::getNMIRs, MIPCallbackI::getNnodes, MIPCallbackI::getNremainingNodes, MIPCallbackI::getObjCoef, MIPCallbackI::getObjCoef, MIPCallbackI::getObjCoefs, MIPCallbackI::getObjCoefs, MIPCallbackI::getPriority, MIPCallbackI::getPriority, MIPCallbackI::getUserThreads, MIPCallbackI::hasIncumbent

Inherited methods from IloCplex::CallbackI

CallbackI::abort, CallbackI::duplicateCallback, CallbackI::getEnv, CallbackI::getModel, CallbackI::getNcols, CallbackI::getNQCs, CallbackI::getNrows, CallbackI::main

Note: This is an advanced routine. Advanced routines typically demand a profound understanding of the algorithms used by ILOG CPLEX. Thus they incur a higher risk of incorrect behavior in your application, behavior that can be difficult to debug. Therefore, ILOG encourages you to consider carefully whether you can accomplish the same task by means of other Callable Library routines instead.
## Index

<table>
<thead>
<tr>
<th>A</th>
<th>Algorithm 76</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>BarrierCallbackI 78</td>
</tr>
<tr>
<td></td>
<td>BasisStatus 80</td>
</tr>
<tr>
<td></td>
<td>BasisStatusArray 81</td>
</tr>
<tr>
<td></td>
<td>BoolParam 82</td>
</tr>
<tr>
<td></td>
<td>BranchCallbackI 84</td>
</tr>
<tr>
<td></td>
<td>BranchDirection 94</td>
</tr>
<tr>
<td></td>
<td>BranchDirectionArray 95</td>
</tr>
<tr>
<td></td>
<td>BranchType 93, 148</td>
</tr>
<tr>
<td>C</td>
<td>Callback 96</td>
</tr>
<tr>
<td></td>
<td>CallbackI 98</td>
</tr>
<tr>
<td></td>
<td>ContinuousCallbackI 103</td>
</tr>
<tr>
<td></td>
<td>ControlCallbackI 106</td>
</tr>
<tr>
<td></td>
<td>CplexStatus 115</td>
</tr>
<tr>
<td></td>
<td>CrossoverCallbackI 118</td>
</tr>
<tr>
<td></td>
<td>CutCallbackI 120</td>
</tr>
<tr>
<td>D</td>
<td>DeleteMode 123</td>
</tr>
<tr>
<td></td>
<td>DisjunctiveCutCallbackI 124</td>
</tr>
<tr>
<td></td>
<td>DualPricing 126</td>
</tr>
<tr>
<td>E</td>
<td>Exception 127</td>
</tr>
<tr>
<td>F</td>
<td>FlowMIRCutCallbackI 128</td>
</tr>
<tr>
<td></td>
<td>FractionalCutCallbackI 130</td>
</tr>
<tr>
<td>G</td>
<td>Goal 132</td>
</tr>
<tr>
<td></td>
<td>GoalI 134</td>
</tr>
<tr>
<td>H</td>
<td>HeuristicCallbackI 151</td>
</tr>
<tr>
<td>I</td>
<td>IISSStatus 157</td>
</tr>
<tr>
<td></td>
<td>IISStatusArray 158</td>
</tr>
<tr>
<td></td>
<td>ILOBARRIERCALLBACK0 18</td>
</tr>
<tr>
<td></td>
<td>ILOBRANCHCALLBACK0 19</td>
</tr>
<tr>
<td></td>
<td>ILOCONTINUOUSCALLBACK0 20</td>
</tr>
<tr>
<td></td>
<td>IloCplex 38</td>
</tr>
<tr>
<td></td>
<td>ILOCPLEXGOAL0 21</td>
</tr>
<tr>
<td></td>
<td>ILOCROSSOVERCALLBACK0 23</td>
</tr>
<tr>
<td></td>
<td>ILOCUTCALLBACK0 24</td>
</tr>
<tr>
<td></td>
<td>ILODISJUNCTIVECUTCALLBACK0 25</td>
</tr>
</tbody>
</table>
INDEX

ILOGCPLEXC++API9.0REFERENCEMANUAL

ILOFRACTIONALCUTCALLBACK0 26
ILOHEURISTICCUTCALLBACK0 27
ILOINCUMBENTCUTCALLBACK0 28
ILOLAZYCONSTRAINTCUTCALLBACK0 29
ILOMIPCALLBACK0 30
ILONETWORKCUTCALLBACK0 31
ILONODECALLBACK0 32
ILOPRESOLVECALLBACK0 33
ILOPROBINGCUTCALLBACK0 34
ILOSIMPLEXCUTCALLBACK0 35
ILOUSERCUTCALLBACK0 37
IncumbentCallbackI 159
IntegerFeasibility 114, 149
IntegerFeasibilityArray 150
IntParam 162
InvalidCutException 170

L
LazyConstraintCallbackI 171, 233

M
MIPCallbackI 173
MIPEmphasisType 181
MultipleConversionException 183
MultipleObjException 184

N
NetworkCallbackI 185
NodeCallbackI 187
NodeData 180
NodeEvaluator 192
NodeEvaluatorI 194
NodeSelect 198
NumParam 199

O
optim.cplex.advanced.cpp 231
optim.cplex.cpp 12
optim.cplex.cpp.logical 230
optim.cplex.cpp.piecewiselinear 228

P
PresolveCallbackI 203
PrimalPricing 205
ProbingCallbackI 206
PWLFormulation 202, 230

Q
Quality 209

S
SearchLimit 213
SearchLimitI 215
SimplexCallbackI 217
SolveCallbackI 219
StringParam 223

T
Type 102

U
UnknownExtractableException 224
UserCutCallbackI 225, 235

V
VariableSelect 227