ILOG CPLEX 8.1

Getting Started

December 2002

Table of Contents

Preface **Introducing ILOG CPLEX** 11

What Is ILOG CPLEX? 12
ILOG CPLEX Technologies 13
Optimizer Options 13
Data Entry Options 14
Solving an LP with CPLEX Technology 15
Using the Interactive Optimizer 15
Concert Technology for C++ Users 16
Concert Technology for Java Users 18
Using the Callable Library 18
What You Need to Know 21
Manual Organization 21
Notation in this Manual 21
Related Documentation 22
For More Information 23
Customer Support 23
Web Site 23

Chapter 1 **Setting Up CPLEX** 25

Installing CPLEX 26
Setting Up Licensing 28

Using the Component Libraries 28

Chapter 2 Interactive Optimizer Tutorial 31

Starting ILOG CPLEX 32

Using Help 32

Entering a Problem 34

Entering the Example Problem 34

Using the LP Format 35

Entering Data 37

Displaying a Problem. 38

Displaying Problem Statistics. 39

Specifying Item Ranges 40

Displaying Variable or Constraint Names 40

Ordering Variables 41

Displaying Constraints 41

Displaying the Objective Function 41

Displaying Bounds 42

Solving a Problem 42

Solving the Example Problem 42

Solution Options. 44

Displaying Post-Solution Information 45

Performing Sensitivity Analysis 46

Writing Problem and Solution Files 47

Selecting a Write File Format. 48

Writing LP Files 49

Writing Basis Files 50

Using Path Names 50

Reading Problem Files 51

Selecting a Read File Format. 51

Reading LP Files 51

Using File Extensions. 52

Reading MPS Files 52

Reading Basis Files 53
Setting ILOG CPLEX Parameters 54
Adding Constraints and Bounds 55
Changing a Problem 56
Changing Constraint or Variable Names 57
Changing Sense. 57
Changing Bounds. 58
Removing Bounds 58
Changing Coefficients 58
Deleting 59
Executing Operating System Commands 60
Quitting ILOG CPLEX 61

Chapter 3 Concert Technology Tutorial for C++ Users 63

The Design of CPLEX in Concert Technology 64
Compiling and Linking CPLEX in Concert Technology Applications. 65
Testing Your Installation on UNIX 65
Testing Your Installation on Windows 65
In Case of Problems 66
The Anatomy of a Concert Technology Application 66
Constructing the Environment — IloEnv 66
Creating a Model — IloModel. 67
Solving the Model — IloCplex 69
Querying Results 70
Handling Errors 71
Building and Solving a Small LP Model in C++. 72
General Structure of a CPLEX Concert Technology Application 72
Modeling by Rows 73
Modeling by Columns. 73
Modeling by Nonzero Elements 74
Complete Program. 74
Writing and Reading Models and Files 79

Selecting an Optimizer 80
Reading a Problem from a File: Example ilolpex2.cpp. 80
Reading the Model from a File 81
Selecting the Optimizer 81
Accessing Basis Information 81
Querying Quality Measures 82
Complete Program 82
Modifying and Reoptimizing 84
Modifying an Optimization Problem: Example ilolpex3.cpp 85
Setting CPLEX Parameters 86
Modifying an Optimization Problem 86
Starting from a Previous Basis 87
Complete Program 87

Chapter 4 Concert Technology for Java Users 89

Compiling CPLEX Applications in Concert Technology 89
In Case Problems Arise 90
The Design of CPLEX in Concert Technology 91
The Anatomy of a Concert Technology Application. 92
Create the Model 93
Solve the Model 94
Query the Results 95
Building and Solving a Small LP Model in Java 95
Modeling by Rows 97
Modeling by Columns. 97
Modeling by Nonzeros 98
Complete Code of LPex1.java. 98

Chapter 5 Callable Library Tutorial. 103

The Design of the ILOG CPLEX Callable Library 103
Compiling and Linking Callable Library Applications 104
Building CPLEX Callable Library Applications on UNIX Platforms 105

Building CPLEX Callable Library Applications on Win32 Platforms 105
Building Applications that Use the CPLEX Parallel Optimizers 106
How ILOG CPLEX Works 106
Opening the ILOG CPLEX Environment 106
Instantiating the Problem Object 107
Populating the Problem Object 107
Changing the Problem Object 107
Creating a Successful Callable Library Application 108
Prototype the Model 108
Identify the Routines to be Called 108
Test Procedures in the Application 109
Assemble the Data 109
Choose an Optimizer 110
Observe Good Programming Practices 110
Debug Your Program 110
Test Your Application 111
Use the Examples 111
Building and Solving a Small LP Model in C 111
Complete Program 113
Reading a Problem from a File: Example lpex2.c 122
Complete Program 124
Adding Rows to a Problem: Example lpex3.c 131
Complete Program 132
Performing Sensitivity Analysis 138

Index 141

Introducing ILOG CPLEX

This preface introduces ILOG CPLEX 8.1. It includes sections on:

- What Is ILOG CPLEX?
- Solving an LP with CPLEX Technology
- What You Need to Know
- Manual Organization
- Notation in this Manual
- Related Documentation
- For More Information

What Is ILOG CPLEX?

ILOG CPLEX is a tool for solving linear optimization problems, commonly referred to as Linear Programming (LP) problems, of the form:

Maximize (or Minimize)	$c_1x_1 + c_2x_2 + ... + c_nx_n$
subject to	$a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n \sim b_1$ $a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n \sim b_2$ ... $a_{m1}x_1 + a_{m2}x_2 + ... + a_{mn}x_n \sim b_m$
with these bounds	$l_1 \leq x_1 \leq u_1$ ... $l_n \leq x_n \leq u_n$

where ~ can be $\leq$, $\geq$, or =, and the upper bounds u_i and lower bounds l_i may be positive infinity, negative infinity, or any real number.

The elements of data you provide as input for this LP are:

Objective function coefficients	$c_1, c_2, ..., c_n$
Constraint coefficients	$a_{11}, a_{21}, ..., a_{n1}$ ... $a_{m1}, a_{m2}, ..., a_{mn}$
Right-hand sides	$b_1, b_2, ..., b_m$
Upper and lower bounds	$u_1, u_2, ..., u_n$ and $l_1, l_2, ..., l_n$

The optimal solution that CPLEX computes and returns is:

Variables	$x_1, x_2, ..., x_n$

CPLEX also can solve several extensions to LP:

- Network Flow problems, a special case of LP that CPLEX can solve much faster by exploiting the problem structure.
- Quadratic Programming (QP) problems, where the LP objective function is expanded to include quadratic terms.
- Mixed Integer Programming (MIP) problems, where any or all of the LP or QP variables are further restricted to take integer values in the optimal solution (and where MIP itself is extended to include constructs like Special Ordered Sets (SOS) and semi-continuous variables).

ILOG CPLEX Technologies

CPLEX comes in three forms to meet a wide range of users' needs:

- The **CPLEX Interactive Optimizer** is an executable program that can read a problem interactively or from files in certain standard formats, solve the problem, and deliver the solution interactively or into text files. The program consists of the file `cplex.exe` on Windows platforms or `cplex` on UNIX platforms.
- **Concert Technology** is a set of C++ and Java class libraries offering an API that includes modeling facilities to allow the programmer to embed CPLEX optimizers in C++ or Java applications. The library is provided in the files in the table below.

Table 1: *Concert Technologies Class Libraries*

	MS Windows	Unix
C++	`ilocplex.lib` `concert.lib`	`libilocplex.a` `libconcert.a`
Java	`cplex.jar`	`cplex.jar`

The Concert Technology libraries make use of the Callable Library (described next).

- The **CPLEX Callable Library** is a C library that allows the programmer to embed CPLEX optimizers in applications written in C, Visual Basic, FORTRAN, or any other language that can call C functions.The library is provided in files `cplex81.lib` and `cplex81.dll` on Windows platforms, and in `libcplex.a`, `libcplex81.so`, and/or `libcplex81.sl` on UNIX platforms.

In this manual, the phrase "CPLEX Component Libraries" is used when referring equally to any of these libraries. While all of the libraries are callable, the term "CPLEX Callable Library" as used here refers specifically to the C library.

Compatible Platforms

ILOG CPLEX is available on Windows and UNIX platforms. The programming interface works the same way and provides the same facilities on all platforms.

Installation Requirements

If you have not yet installed ILOG CPLEX on your platform, please consult Chapter 1, *Setting Up CPLEX*. It contains instructions for installing ILOG CPLEX.

Optimizer Options

This manual explains how to use all the LP, QP and MIP algorithms that are part of ILOG CPLEX. Some users may not have access to all algorithms. Such users should consult their ILOG account manager or the ILOG support web site to determine to which algorithms they have access.

Default settings will result in a call to an optimizer that is appropriate to the class of problem you are solving. However you may wish to choose a different optimizer for special purposes. An LP or QP problem can be solved using any of the following CPLEX optimizers: Dual Simplex, Primal Simplex, Barrier, and perhaps also the Network Optimizer (if the problem contains an extractable network substructure). Pure network models are all solved by the Network Optimizer. MIP models are all solved by the Mixed Integer Optimizer, which in turn may invoke any of the LP or QP optimizers in the course of its computation. Table 2 summarizes these possible choices.

Table 2: *Optimizers*

	LP	Network	QP	MIP
Dual Optimizer	➨		➨	
Primal Optimizer	➨		➨	
Barrier Optimizer	➨		➨	
Mixed Integer Optimizer				➨
Network Optimizer	Note 1	➨	Note 1	
Note 1: The problem must contain an extractable network substructure.				

The choice of optimizer or other parameter settings may have a very large effect on the solution speed of your particular class of problem. The *ILOG CPLEX User's Manual* describes the optimizers, provides suggestions for maximizing performance, and notes the features and algorithmic parameters unique to each optimizer.

Using the Parallel Optimizers

On a computer with multiple CPUs, the Barrier Optimizer and the MIP Optimizer are each capable of running in parallel, that is, they can apply these additional CPUs to the task of optimizing the model. The number of CPUs used by an optimizer is controlled by the user; under default settings these optimizers run in serial (single CPU) mode. When solving small models, such as those described in this document, the effect of parallelism will generally be negligible. On larger models, the effect is ordinarily beneficial to solution speed. See the section *Using Parallel Optimizers* in the *ILOG CPLEX User's Manual* for information on using CPLEX on a parallel computer.

Data Entry Options

CPLEX provides several options for entering your problem data. When using the Interactive Optimizer, most users will enter problem data from formatted files. CPLEX supports the industry-standard MPS (Mathematical Programming System) file format as well as CPLEX LP format, a row-oriented format many users may find more natural. Interactive entry (using CPLEX LP format) is also a possibility for small problems.

Data entry options are described briefly in this manual. File formats are documented in an appendix of the *ILOG CPLEX Reference Manual*.

Concert Technology and Callable Library users may read problem data from the same kinds of files as in the Interactive Optimizer, or they may want to pass data directly into CPLEX to gain efficiency. These options are discussed in a series of examples that begin with *Building and Solving a Small LP Model in C++*, *Building and Solving a Small LP Model in Java*, and *Building and Solving a Small LP Model in C* for the CPLEX Callable Library users.

Solving an LP with CPLEX Technology

To help you learn which CPLEX technology best meets your needs, we briefly demonstrate here how to create and solve an LP model, using four different interfaces to CPLEX. Full details of writing a practical program are in the chapters containing the tutorials.

The problem to be solved is:

Maximize	$x_1 + 2x_2 + 3x_3$
subject to	$-x_1 + x_2 + x_3 \leq 20$ $x_1 - 3x_2 + x_3 \leq 30$
with these bounds	$0 \leq x_1 \leq 40$ $0 \leq x_2 \leq +\infty$ $0 \leq x_3 \leq +\infty$

Using the Interactive Optimizer

The following is screen output from a CPLEX Interactive Optimizer session where the example model is entered and solved. `CPLEX>` indicates the CPLEX prompt, and text following this is user input.

```
Welcome to CPLEX Interactive Optimizer 8.1.0
  with Simplex, Mixed Integer & Barrier Optimizers
Copyright (c) ILOG 1997-2002
CPLEX is a registered trademark of ILOG

Type 'help' for a list of available commands.
Type 'help' followed by a command name for more
information on commands.

CPLEX> enter example
Enter new problem ['end' on a separate line terminates]:
maximize  x1 + 2 x2 + 3 x3
subject to -x1 +  x2 + x3 <= 20
           x1 - 3 x2 + x3 <=30
bounds
```

```
0 <= x1 <= 40
0 <= x2
0 <= x3
end
CPLEX> optimize
Tried aggregator 1 time.
No LP presolve or aggregator reductions.
Presolve time =    0.00 sec.

Iteration log . . .
Iteration:     1   Dual infeasibility =             0.000000
Iteration:     2   Dual objective     =           202.500000

Dual simplex - Optimal:  Objective =   2.0250000000e+002
Solution time =    0.01 sec.  Iterations = 2 (1)

CPLEX> quit
```

Concert Technology for C++ Users

Here is a C++ program using CPLEX in Concert Technology to solve the example model. An expanded version of this example is discussed in detail in Chapter 3, *Concert Technology Tutorial for C++ Users*.

```
#include <ilcplex/ilocplex.h>
ILOSTLBEGIN

int
main (int argc, char **argv)
{
   IloEnv env;
   try {
      IloModel model(env);
      IloNumVarArray x(env);
      x.add(IloNumVar(env, 0.0, 40.0));
      x.add(IloNumVar(env));
      x.add(IloNumVar(env));
      model.add(IloMaximize(env, x[0] + 2 * x[1] + 3 * x[2]));
      model.add( - x[0] +     x[1] + x[2] <= 20);
      model.add(   x[0] - 3 * x[1] + x[2] <= 30);

      IloCplex cplex(model);
      if ( !cplex.solve() ) {
         env.error() << "Failed to optimize LP." << endl;
         throw(-1);
      }

      IloNumArray vals(env);
      env.out() << "Solution status = " << cplex.getStatus() << endl;
      env.out() << "Solution value = " << cplex.getObjValue() << endl;
      cplex.getValues(vals, x);
```

```
      env.out() << "Values = " << vals << endl;
   }
   catch (IloException& e) {
      cerr << "Concert exception caught: " << e << endl;
   }
   catch (...) {
      cerr << "Unknown exception caught" << endl;
   }

   env.end();

   return 0;
}  // END main
```

Concert Technology for Java Users

Here is a Java program using ILOG Concert Technology to solve the example model. An expanded version of this example is discussed in detail in Chapter 4, *Concert Technology for Java Users*.

```
import ilog.concert.*;
import ilog.cplex.*;

public class Example {
  public static void main(String[] args) {
    try {
      IloCplex cplex = new IloCplex();

      double[]    lb = {0.0, 0.0, 0.0};
      double[]    ub = {40.0, Double.MAX_VALUE, Double.MAX_VALUE};
      IloNumVar[] x  = cplex.numVarArray(3, lb, ub);

      double[] objvals = {1.0, 2.0, 3.0};
      cplex.addMaximize(cplex.scalProd(x, objvals));

      cplex.addLe(cplex.sum(cplex.prod(-1.0, x[0]),
                            cplex.prod( 1.0, x[1]),
                            cplex.prod( 1.0, x[2])), 20.0);
      cplex.addLe(cplex.sum(cplex.prod( 1.0, x[0]),
                            cplex.prod(-3.0, x[1]),
                            cplex.prod( 1.0, x[2])), 30.0);

      if ( cplex.solve() ) {
        cplex.out().println("Solution status = " + cplex.getStatus());
        cplex.out().println("Solution value  = " + cplex.getObjValue());

        double[] val = cplex.getValues(x);
        int ncols = cplex.getNcols();
        for (int j = 0; j < ncols; ++j)
           cplex.out().println("Column: " + j + " Value = " + val[j]);
      }
      cplex.end();
    }
    catch (IloException e) {
      System.err.println("Concert exception '" + e + "' caught");
    }
  }
}
```

Using the Callable Library

Here is a C program using the CPLEX Callable Library to solve the example model. An expanded version of this example is discussed in detail in Chapter 5, *Callable Library Tutorial*.

```
#include <ilcplex/cplex.h>
#include <stdlib.h>
```

```
#include <string.h>

#define NUMROWS    2
#define NUMCOLS    3
#define NUMNZ      6

int
main (int argc, char **argv)
{
   int       status = 0;
   CPXENVptr env = NULL;
   CPXLPptr  lp  = NULL;

   double   obj[NUMCOLS];
   double   lb[NUMCOLS];
   double   ub[NUMCOLS];
   double   x[NUMCOLS];
   int      rmatbeg[NUMROWS];
   int      rmatind[NUMNZ];
   double   rmatval[NUMNZ];
   double   rhs[NUMROWS];
   char     sense[NUMROWS];

   int      solstat;
   double   objval;

   env = CPXopenCPLEX (&status);
   if ( env == NULL ) {
      char  errmsg[1024];
      fprintf (stderr, "Could not open CPLEX environment.\n");
      CPXgeterrorstring (env, status, errmsg);
      fprintf (stderr, "%s", errmsg);
      goto TERMINATE;
   }

   lp = CPXcreateprob (env, &status, "lpex1");
   if ( lp == NULL ) {
      fprintf (stderr, "Failed to create LP.\n");
      goto TERMINATE;
   }

   CPXchgobjsen (env, lp, CPX_MAX);

       obj[0] = 1.0;      obj[1] = 2.0;           obj[2] = 3.0;
        lb[0] = 0.0;       lb[1] = 0.0;           lb[2]  = 0.0;
        ub[0] = 40.0;      ub[1] = CPX_INFBOUND;  ub[2]  = CPX_INFBOUND;

   status = CPXnewcols (env, lp, NUMCOLS, obj, lb, ub, NULL, NULL);
   if ( status ) {
      fprintf (stderr, "Failed to populate problem.\n");
      goto TERMINATE;
   }
```

```
   rmatbeg[0] = 0;
   rmatind[0] = 0;     rmatind[1] = 1;    rmatind[2] = 2;  sense[0] = 'L';
   rmatval[0] = -1.0;  rmatval[1] = 1.0;  rmatval[2] = 1.0;  rhs[0] = 20.0;

   rmatbeg[1] = 3;
   rmatind[3] = 0;    rmatind[4] = 1;    rmatind[5] = 2;     sense[1] = 'L';
   rmatval[3] = 1.0;  rmatval[4] = -3.0; rmatval[5] = 1.0;   rhs[1]   = 30.0;

   status = CPXaddrows (env, lp, 0, NUMROWS, NUMNZ, rhs, sense, rmatbeg,
                        rmatind, rmatval, NULL, NULL);
   if ( status ) {
      fprintf (stderr, "Failed to populate problem.\n");
      goto TERMINATE;
   }

   status = CPXlpopt (env, lp);
   if ( status ) {
      fprintf (stderr, "Failed to optimize LP.\n");
      goto TERMINATE;
   }

   status = CPXsolution (env, lp, &solstat, &objval, x, NULL, NULL, NULL);
   if ( status ) {
      fprintf (stderr, "Failed to obtain solution.\n");
      goto TERMINATE;
   }
   printf ("\nSolution status = %d\n", solstat);
   printf ("Solution value  = %f\n", objval);
   printf ("Solution        = [%f, %f, %f]\n\n", x[0], x[1], x[2]);

TERMINATE:

   if ( lp != NULL ) {
      status = CPXfreeprob (env, &lp);
      if ( status ) {
         fprintf (stderr, "CPXfreeprob failed, error code %d.\n", status);
      }
   }

   if ( env != NULL ) {
      status = CPXcloseCPLEX (&env);
      if ( status ) {
         char  errmsg[1024];
         fprintf (stderr, "Could not close CPLEX environment.\n");
         CPXgeterrorstring (env, status, errmsg);
         fprintf (stderr, "%s", errmsg);
      }
   }

   return (status);

}  /* END main */
```

What You Need to Know

In order to use ILOG CPLEX effectively, you need to be familiar with your operating system, whether Unix or Windows.

This manual assumes you already know how to create and manage files. In addition, if you are building an application that uses the Component Libraries, this manual assumes that you know how to compile, link, and execute programs written in a high-level language. The Callable Library is written in the C programming language, while Concert Technology is written in C++ and Java. This manual also assumes that you already know how to program in the appropriate language and that you will consult a programming guide when you have questions in that area.

Manual Organization

Chapter 1, *Setting Up CPLEX* tells how to install CPLEX.

Chapter 2, *Interactive Optimizer Tutorial*, describes, step by step, how to use the Interactive Optimizer—how to start it, how to enter problems and data, how to read and save files, how to modify objective functions and constraints, and how to display solutions and analytical information.

Chapter 3, *Concert Technology Tutorial for C++ Users* describes the same activities using the classes in the C++ version of the CPLEX Concert Technology Library.

Chapter 4, *Concert Technology for Java Users* describes the same activities using the classes in the Java version of the CPLEX Concert Technology Library.

Chapter 5, *Callable Library Tutorial* describes the same activities using the routines in the ILOG CPLEX Callable Library.

All tutorials use examples that are delivered with the standard distribution.

Notation in this Manual

To make this manual easier to use, we've followed a few conventions in notation and names.

- Important ideas are *italicized* the first time they appear.
- Text that is entered at the keyboard or displayed on the screen and commands and their options available through the Interactive Optimizer appear in `this typeface`, for example, `set preprocessing aggregator n`.
- Entries that you must fill in appear in *`this typeface`*; for example, `write` *`filename`*.

- The names of C routines and parameters in the ILOG CPLEX Callable Library begin with `CPX`; the names of C++ and Java classes in the CPLEX Concert Technology Library begin with `Ilo`; and both appear in `this typeface`, for example: `CPXcopyobjnames()` or `IloCplex`.
- Combinations of keys from the keyboard are hyphenated. For example, control-c indicates that you should press the control key and the c key simultaneously. The `<return>` indicates end of line or end of data entry. On some keyboards, the key is labeled enter or Enter.

Related Documentation

In addition to this introductory manual, the standard distribution of ILOG CPLEX comes with the *ILOG CPLEX User's Manual*, the *ILOG CPLEX Reference Manual*, and the *ILOG Concert Technology Documentation Kit*. All ILOG documentation is available in an online version in HTML (hypertext mark-up language). It is delivered with the standard distribution of the product and accessible through conventional HTML browsers.

- The *ILOG CPLEX User's Manual* explains the relationship between the Interactive Optimizer and the Component Libraries. It enlarges on aspects of linear programming with ILOG CPLEX and shows you how to handle quadratic programming (QP) problems and mixed integer programming (MIP) problems. It tells you how to control ILOG CPLEX parameters, debug your applications, and efficiently manage input and output. It also explains how to use parallel CPLEX optimizers.
- The *ILOG CPLEX Reference Manual* documents the Callable Library routines and their arguments, the Concert Technology classes, methods, and functions, and the commands and options of the Interactive Optimizer. The reference manual also contains a table of parameters that can be modified by parameter routines, a list of error messages, and details about file formats.
- The *ILOG CPLEX Java Reference Manual* supplies detailed definitions of the Concert Technology interfaces and CPLEX Java classes. It is available only in online form as HTML, and Microsoft compiled HTML help (.CHM) form.
- The *ILOG Concert Technology Documentation Kit* includes the *ILOG Concert Technology Reference Manual*, which documents the classes, methods, and functions of the Concert Technology library; the *ILOG Concert Technology User's Manual*, which provides examples that show how to use Concert Technology to model problems; the *ILOG Hybrid Cooperating Optimizers User's Guide & Reference*, which documents the class `IloLinConstraint` and shows how to use ILOG's main algorithm classes, `IloSolver` and `IloCplex` in cooperation; and the *ILOG Concert Technology Migration Guide*, which shows how to translate applications created in previous versions of ILOG products to Concert Technology.

As you work with ILOG CPLEX on a long-term basis, you should read the complete *User's Manual* to learn how to design models and implement solutions to your own problems. Consult the *ILOG CPLEX Reference Manual* for authoritative documentation of the Component Libraries and Interactive Optimizer.

For More Information

ILOG offers technical support and comprehensive Web sites for its products.

Customer Support

For technical support of ILOG CPLEX, you should contact your local distributor, or, if you are a direct ILOG customer, contact the nearest regional office:

Region	E-mail	Telephone	Fax
France	cplex-support@ilog.fr	0 800 09 27 91 (numéro vert) +33 (0)1 49 08 35 62	+33 (0)1 49 08 35 10
Germany	cplex-support@ilog.de	+49 6172 40 60 33	+49 6172 40 60 10
Spain	cplex-support@ilog.es	+34 91 710 2480	+34 91 372 9976
United Kingdom	cplex-support@ilog.co.uk	+44 (0)1344 661600	+44 (0)1344 661601
Other European countries	cplex-support@ilog.fr	+33 (0)1 49 08 35 62	+33 (0)1 49 08 35 10
Japan	cplex-support@ilog.co.jp	+81 3 5211 5770	+81 3 5211 5771
Singapore	cplex-support@ilog.com.sg	+65 6773 06 26	+65 6773 04 39
USA	cplex-support@ilog.com	1-877-ILOG-TECH 1-877-456-4832 (toll free) or 1-650-567-8080	+1 650 567 8001

We encourage you to use e-mail for faster, better service.

Web Site

The CPLEX Web site at `http://www.ilog.com/products/cplex/` offers product descriptions, press releases, and contact information. It lists services, such as training, maintenance, technical support, and outlines special programs. In addition, it links you to an `ftp` site where you can pick up examples.

The technical support pages contain FAQ (Frequently Asked/Answered Questions) and the latest patches for some of our products. Changes are posted in the product mailing list. Access to these pages is restricted to owners of an ongoing maintenance contract. The maintenance contract number and the name of the person this contract is sent to in your company will be needed for access, as explained on the login page.

All three of the following sites contain the same information, but access is localized, so we recommend that you connect to the site corresponding to your location and select the "support" page from the home page.

- The Americas: `http://www.ilog.com`
- Asia & Pacific nations: `http://www.ilog.com.sg`
- Europe, Africa, and Middle East: `http://www.ilog.fr`

On those Web pages, you will find additional information about ILOG CPLEX in technical papers that have also appeared at industrial and academic conferences.

1

Setting Up CPLEX

You install ILOG CPLEX in two steps: first, transfer the files from the distribution medium (a CD or an FTP site) into a directory on your local file system; then activate your license.

At that point, all of the features of CPLEX become functional and are available to you. The chapters that follow this one provide tutorials in the use of each of the Technologies that CPLEX provides: the Concert Technology Tutorials for C++ and Java users, and the Callable Library Tutorial for C and other languages.

This chapter provides guidelines for:

- Installing CPLEX
- Setting Up Licensing
- Using the Component Libraries

Important: Please read these instructions in their entirety before beginning the installation. Remember that most CPLEX distributions will operate correctly only on the specific platform and operating system version for which they are designed. If you upgrade your operating system, you may need to obtain a new CPLEX distribution.

Installing CPLEX

The steps to perform CPLEX installation involve identifying the correct distribution file for your particular platform, and then executing a command that uses that distribution file. The identification step is described in the booklet that comes with the CD-ROM, or is provided with the FTP instructions for download. Once the correct distribution file is at hand, the installation proceeds as follows.

Installation on UNIX

On UNIX systems CPLEX 8.1 is installed in a subdirectory named `cplex81`, under the current working directory where you perform the installation.

Use the `cd` command to move to the top level directory into which you wish to install the CPLEX subdirectory. Then type this command:

```
gzip -dc < path/cplex.tgz | tar xf -
```

where `path` is the full path name pointing to the location of the CPLEX distribution file (either on the CD-ROM or on a disk where you performed the FTP download). On UNIX systems, both CPLEX and Concert are installed when you execute the above command.

Installation on Windows

Before you install CPLEX, you need to identify the correct distribution file for your platform. There are instructions on how to identify your distribution in the booklet that comes with the CD-ROM or with the FTP instructions for download. This booklet also describes how to start the CPLEX installation on your platform.

Directory Structure

After completing the installation, you will have a directory structure like the following:

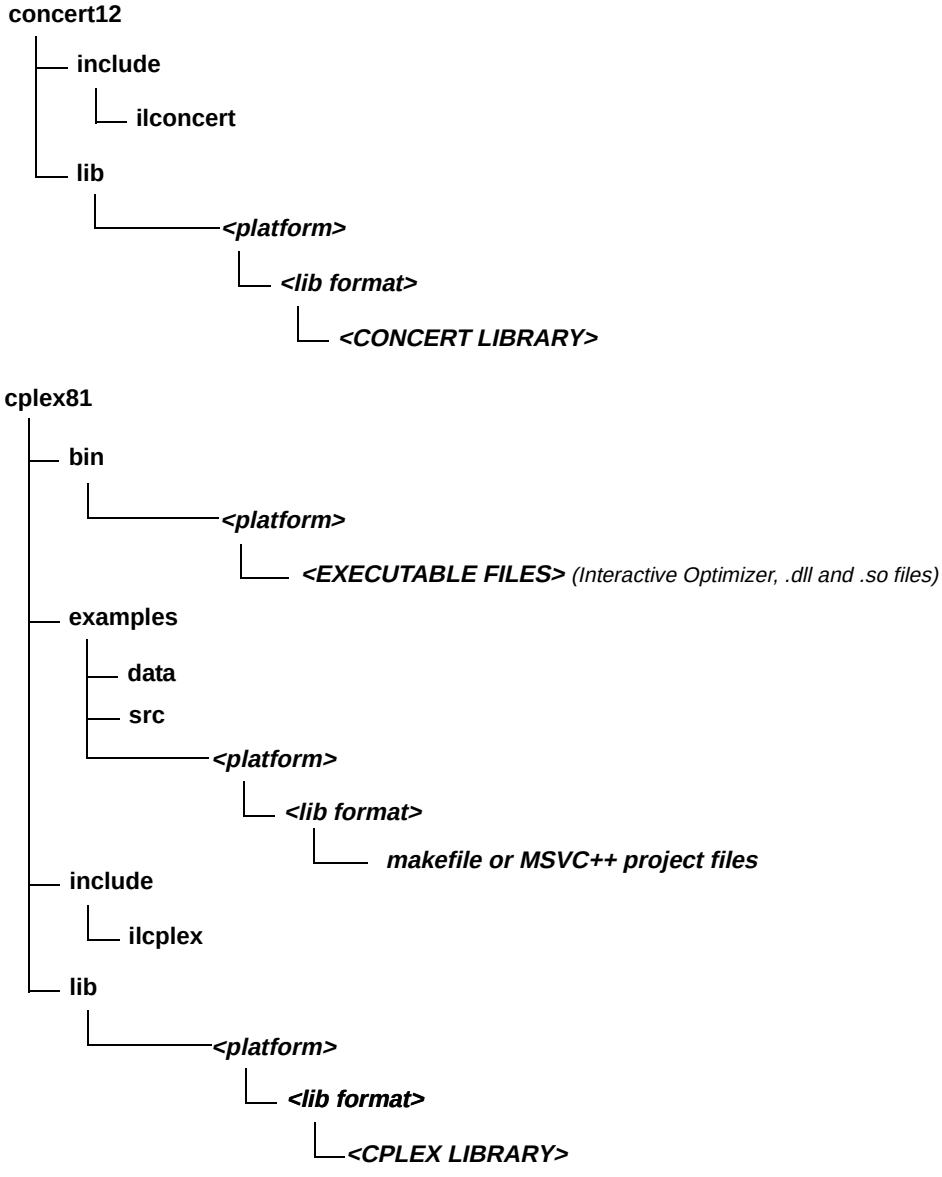

Figure 1.0 *Installation Directory Structures*

Be sure to read the `readme.html` carefully for the latest information on the version of CPLEX you have installed.

Setting Up Licensing

CPLEX 8.1 runs under the control of the ILOG License Manager (ILM). Before you can run CPLEX, or any application that calls it, you must have established a valid license that ILM can read. Licensing instructions are provided in the *ILOG License Manager User's Guide & Reference*, which is included with the standard CPLEX product distribution. The basic steps are:

1. Install ILM. Normally you obtain ILM distribution media from the same place that you obtain CPLEX.
2. Run the `ihostid` program, which is found in the directory where you install ILM.
3. Communicate the output of step 2 to your local ILOG sales administration department. They will send you a license key in return.
4. Create a file on your system to hold this license key, and set the environment variable `ILOG_LICENSE_FILE` so that CPLEX will know where to find the license key. (The environment variable need not be used if you install the license key in a platform-dependent default file location.)

Using the Component Libraries

After you have completed the installation and licensing steps, you can verify that everything is working by running one or more of the examples that are provided with the standard distribution.

Verifying Installation on UNIX

On a UNIX system, go to the subdirectory `examples/<machine>/<libformat>` that matches your particular platform, and in it you will find a file named `makefile`. Execute one of the examples, for instance `lpex1.c`, by doing

```
make lpex1
```

```
lpex1 -r    # this example takes one argument, either -r, -c, or -n
```

If your interest is in running one of the C++ examples, try

```
make ilolpex1
```

```
ilolpex1 -r # this is the same as lpex1 and takes the same arguments.
```

If your interest is in running one of the Java examples, try

```
make LPex1.class
java -Djava.library.path=../../../bin/<platform>:
     -classpath ../../../lib/cplex.jar: LPex1 -r
```

Any of these examples should return an optimal objective function value of 202.5.

Verifying Installation on Windows

On a Windows machine, you can follow a similar process using the facilities of your compiler interface to compile and then run any of the examples. A project file for each example is provided, in a format for Microsoft Visual C++ 6 and Visual C++ .NET.

In Case of Errors

If an error occurs during the `make` or compile step, then check that you are able to access the compiler and the necessary linker/loader files and system libraries. If an error occurs on the next step, when executing the program created by `make`, then the nature of the error message will guide your actions. If the problem is in licensing, consult the *ILOG License Manager User's Guide and Reference* for further guidance. For Windows users, if the program has trouble locating `cplex81.dll`, make sure the DLL is stored either in the current directory or in a directory listed in your `PATH` environment variable.

The UNIX `makefile`, or Windows project file, contains useful information regarding recommended compiler flags and other settings for compilation and linking.

Compiling and Linking Your Own Applications

The source files for the examples and the makefiles provide guidance for how your own application can call CPLEX. The following chapters give more specific information on the necessary header files for compilation, and how to link CPLEX and Concert Technology library files into your application.

- Chapter 3, *Concert Technology Tutorial for C++ Users* contains information and platform-specific instructions for compiling and linking the Concert Technology Library, for C++ users.
- Chapter 4, *Concert Technology for Java Users* contains information and platform-specific instructions for compiling and linking the Concert Technology Library, for Java users.
- Chapter 5, *Callable Library Tutorial* contains information and platform-specific instructions for compiling and linking the Callable Library.

C H A P T E R

2

Interactive Optimizer Tutorial

This step-by-step tutorial introduces the major features of the ILOG CPLEX Interactive Optimizer. In this chapter, you will learn about:

- Starting ILOG CPLEX
- Using Help
- Entering a Problem
- Displaying a Problem
- Solving a Problem
- Performing Sensitivity Analysis
- Writing Problem and Solution Files
- Reading Problem Files
- Setting ILOG CPLEX Parameters
- Adding Constraints and Bounds
- Changing a Problem
- Executing Operating System Commands
- Quitting ILOG CPLEX

Starting ILOG CPLEX

To start the ILOG CPLEX Interactive Optimizer, at your operating system prompt type the command:

```
cplex
```

A message similar to the following one appears on the screen:

```
Welcome to CPLEX Interactive Optimizer 8.1.0
  with Simplex, Mixed Integer & Barrier Optimizers
Copyright (c) ILOG 1997-2002
CPLEX is a registered trademark of ILOG

Type help for a list of available commands.
Type help followed by a command name for more
information on commands.


CPLEX>
```

The last line, `CPLEX>`, is the prompt, indicating that the product is running and is ready to accept one of the available ILOG CPLEX commands. Use the `help` command to see a list of these commands.

Using Help

ILOG CPLEX accepts commands in several different formats. You can type either the full command name, or any shortened version that uniquely identifies that name. For example, enter `help` after the `CPLEX>` prompt, as shown:

```
CPLEX> help
```

You will see a list of the ILOG CPLEX commands on the screen.

Since all commands start with a unique letter, you could also enter just the single letter `h`.

```
CPLEX> h
```

ILOG CPLEX does not distinguish between upper and lower case letters, so you could enter `h`, `H`, `help`, or `HELP`. All of these variations invoke the `help` command. The same rules apply to all ILOG CPLEX commands. You need only type enough letters of the command to distinguish it from all other commands, and it does not matter whether you type upper and lower case letters. Throughout this manual, we use lower case letters.

After you type the `help` command, a list of available commands with their descriptions appears on the screen, like this:

```
add         add constraints to problem
baropt      solve using barrier algorithm
change      change the problem
display     display problem, solution, or parameter settings
enter       enter a new problem
help        provide information on CPLEX commands
mipopt      solve a mixed integer program
netopt      solve the problem using network method
optimize    solve the problem
primopt     solve using the primal method
quit        leave CPLEX
read        read problem or basis information from a file
set         set parameters
tranopt     solve using the dual method
write       write problem or solution info. to a file
xecute      execute a command from the operating system

Enter enough characters to uniquely identify commands & options.
Commands can be entered partially (CPLEX will prompt you for
further information) or as a whole.
```


To find out more about a specific command, type `help` followed by the name of that command. For example, to learn more about the `primopt` command type:

```
help primopt
```

Typing the full name is unnecessary. Alternatively, you can try:

```
h p
```

The following message appears to tell you more about the use and syntax of the `primopt` command:

```
The PRIMOPT command solves the current problem using
a primal simplex method or crosses over to a basic solution
if a barrier solution exists.

Syntax:  PRIMOPT

A problem must exist in memory (from using either the
ENTER or READ command) in order to use the PRIMOPT
command.

Sensitivity information (dual price and reduced-cost
information) as well as other detailed information about
the solution can be viewed using the DISPLAY command,
after a solution is generated.
```

Summary

The syntax for the `help` command is:

```
help command name
```

Entering a Problem

Most users with larger problems enter problems by reading data from formatted files. That practice is described in *Reading Problem Files* on page 51. For now, let's enter a smaller problem from the keyboard by using the `enter` command. The process is described step-by-step in the topics:

- Entering the Example Problem
- Using the LP Format
- Entering Data

Entering the Example Problem

As an example, we will use the following problem:

Maximize	$x_1 + 2x_2 + 3x_3$
subject to	$-x_1 + x_2 + x_3 \leq 20$ $x_1 - 3x_2 + x_3 \leq 30$
with these bounds	$0 \leq x_1 \leq 40$ $0 \leq x_2 \leq +\infty$ $0 \leq x_3 \leq +\infty$

This problem has three variables (x_1, x_2, and x_3) and two less-than-or-equal-to constraints.

The `enter` command is used to enter a new problem from the keyboard. The procedure is almost as simple as typing the problem on a page. At the `CPLEX>` prompt type:

```
enter
```

A prompt appears on the screen asking you to give a name to the problem that you are about to enter.

Naming a Problem

The problem name may be anything that is allowed as a file name in your operating system. If you decide that you do not want to enter a new problem, just press the `<return>` key without typing anything. The `CPLEX>` prompt will reappear without causing any action. The same can be done at any `CPLEX>` prompt. If you do not want to complete the command, simply press the `<return>` key. For now, type in the name `example` at the prompt.

```
Enter name for problem: example
```

The following message appears:

```
Enter new problem ['end' on a separate line terminates]:
```

and the cursor is positioned on a blank line below it where you can enter the new problem.

You can also type the problem name directly after the `enter` command and avoid the intermediate prompt.

Summary

The syntax for entering a problem is:

```
enter problem name
```

Using the LP Format

Entering a new problem is basically like typing it on a page, but there are a few rules to remember. These rules conform to the ILOG CPLEX LP file format and are documented in the *ILOG CPLEX Reference Manual*. We use LP format throughout this tutorial.

The problem should be entered in the following order:

1. Objective Function
2. Constraints
3. Bounds

Objective Function

Before entering the objective function, you must state whether the problem is a minimization or maximization. For this example, you type:

```
maximize
x1 + 2x2 + 3x3
```

You may type `minimize` or `maximize` on the same line as the objective function, but you must separate them by at least one space.

Variable Names

In the example, the variables are named simply `x1`, `x2`, `x3`, but you can give your variables more meaningful names such as `cars` or `gallons`. The only limitations on variable names in LP format are that the names must be no more than 255 characters long and use only the alphanumeric characters (a-z, A-Z, 0-9) and certain symbols: ! " # $ % & ( ) , . ; ? @ _ ‘ ’ { } ~. Any line with more than 510 characters is truncated.

A variable name cannot begin with a number or a period, and there is one character combination that cannot be used: the letter `e` or `E` alone or followed by a number or another `e`, since this notation is reserved for exponents. Thus, a variable cannot be named `e24` nor `e9cats` nor `eels` nor any other name with this pattern. This restriction applies only to problems entered in LP format.

Constraints

Once you have entered the objective function, you can move on to the constraints. However, before you start entering the constraints, you must indicate that the subsequent lines are constraints by typing:

```
subject to
```

or

```
st
```

These terms can be placed alone on a line or on the same line as the first constraint if separated by at least one space. Now you can type in the constraints in the following way:

```
st
-x1 + x2 + x3 <= 20
x1 - 3x2 + x3 <= 30
```

Constraint Names

In this simple example, it is easy to keep track of the small number of constraints, but for many problems, it may be advantageous to name constraints so that they are easier to identify. You can do so in ILOG CPLEX by typing a constraint name and a colon before the actual constraint. If you do not give the constraints explicit names, ILOG CPLEX will give them the default names `c1, c2, . . . , cn`. In the example, if we want to call the constraints `time` and `labor`, for example, we enter the constraints like this:

```
st
time:  -x1 + x2 + x3 <= 20
labor:  x1 - 3x2 + x3 <= 30
```

Constraint names are subject to the same guidelines as variable names. They must have no more than 16 characters, consist of only allowed characters, and not begin with a number, a period, or the letter `e` followed by a positive or negative number or another `e`.

Objective Function Names

The objective function can be named in the same manner as constraints. The default name for the objective function is `obj`. ILOG CPLEX assigns this name if no other is entered.

Bounds

Finally, you must enter the lower and upper bounds on the variables. If no bounds are specified, ILOG CPLEX will automatically set the lower bound to 0 and the upper bound to $+\infty$. You must explicitly enter bounds only when the bounds differ from the default values. In our example, the lower bound on `x1` is 0, which is the same as the default. The upper bound 40, however, is not the default, so you must enter it explicitly. You must type `bounds` on a separate line before you enter the bound information:

```
bounds
x1 <= 40
```

Since the bounds on x2 and x3 are the same as the default bounds, there is no need to enter them. You have finished entering the problem, so to indicate that the problem is complete, type:

```
end
```

on the last line.

The CPLEX> prompt returns, indicating that you can again enter a ILOG CPLEX command.

Summary

Entering a problem in ILOG CPLEX is straightforward, provided that you observe a few simple rules:

- The terms maximize or minimize must precede the objective function; the term subject to must precede the constraints section; both must be separated from the beginning of each section by at least one space.
- The word bounds must be alone on a line preceding the bounds section.
- On the final line of the problem, end must appear.

Entering Data

You can use the <return> key to split long constraints, and ILOG CPLEX still interprets the multiple lines as a single constraint. When you split a constraint in this way, do not press <return> in the middle of a variable name or coefficient. The following is acceptable:

```
time: -x1 + x2 + <return>
x3 <= 20  <return>
labor:  x1 - 3x2 + x3 <= 30  <return>
```

The entry below, however, is incorrect since the <return> key splits a variable name.

```
time: -x1 + x2 + x <return>
3 <= 20  <return>
labor:  x1 - 3x2 + x3 <= 30  <return>
```

If you type a line that ILOG CPLEX cannot interpret, a message describing the problem will appear, and the entire line will be ignored. You must then re-enter the line.

The final thing to remember when you are entering a problem is that once you have pressed <return>, you can no longer directly edit the characters that precede the <return>. As long as you have not pressed the <return> key, you can use the <backspace> key to go back and change what you typed on that line. Once <return> has been pressed, the change command must be used to modify the problem. The change command is described in *Changing a Problem* on page 56.

Displaying a Problem

Now that you have entered a problem using ILOG CPLEX, you must verify that the problem was entered correctly. To do so, use the `display` command. At the `CPLEX>` prompt type:

```
display
```

A list of the items that can be displayed then appears. Some of the options display parts of the problem description, while others display parts of the problem solution. Options about the problem solution are not available until after the problem has been solved. The list looks like this:

```
Display Options:

iis            display infeasibility diagnostics (IIS constraints)
problem        display problem characteristics
sensitivity    display sensitivity analysis
settings       display parameter settings
solution       display existing solution

Display what:
```

If you type `problem` in reply to that prompt, that option will list a set of problem characteristics, like this:

```
Display Problem Options:

all              display entire problem
binaries         display binary variables
bounds           display a set of bounds
constraints      display a set of constraints or node supply/demand values
generals         display general integer variables
histogram        display a histogram of row or column counts
integers         display integer variables
names            display names of variables or constraints
qpvariables      display quadratic variables
semi-continuous  display semi-continuous and semi-integer variables
sos              display special ordered sets
stats            display problem statistics
variable         display a column of the constraint matrix

Display which problem characteristic:
```

Enter the option `all` to display the entire problem.

```
Maximize
 obj: x1 + 2 x2 + 3 x3
Subject To
 c1: - x1 +   x2 +   x3 <= 20
 c2:  x1 - 3 x2 +   x3 <= 30
Bounds
 0 <= x1 <= 40
All other variables are >= 0.
```

The default names `obj`, `c1`, `c2`, are provided by ILOG CPLEX.

If that is what you want, you are ready to solve the problem. If there is a mistake, you must use the `change` command to modify the problem. The `change` command is described in *Changing a Problem* on page 56.

Summary

Display problem characteristics by entering the command:

```
display problem
```

Displaying Problem Statistics

When the problem is as small as our example, it is easy to display it on the screen; however, many real problems are far too large to display. For these problems, the `stats` option of the `display problem` command is helpful. When you select `stats`, information about the attributes of the problem appears, but not the entire problem itself. These attributes include:

- the number and type of constraints
- variables
- nonzero constraint coefficients

Try this feature by typing:

```
display problem stats
```

For our example, the following information appears:

```
Problem Name: example

Constraints        :    2  [Less: 2]
Variables          :    3  [Nneg: 2,  Box: 1]
Constraint nonzeros:    6
Objective  nonzeros:    3
RHS        nonzeros:    2
```

This information tells us that in the example there are two constraints, three variables, and six nonzero constraint coefficients. The two constraints are both of the type less-than-or-equal-to. Two of the three variables have the default nonnegativity bounds ($0 \leq x \leq +\infty$) and one is restricted to a certain range (a box variable). In addition to a constraint matrix nonzero count, there is a count of nonzero coefficients in the objective function and on the right-hand side. Such statistics can help to identify errors in a problem without displaying it in its entirety.

Another way to avoid displaying an entire problem is to display a specific part of it by using one of the following three options of the `display problem` command:

- `names`, described in *Displaying Variable or Constraint Names* on page 40, can be used to display a specified set of variable or constraint names;

- `constraints`, described in *Displaying Constraints* on page 41, can be used to display a specified set of constraints;
- `bounds`, described in *Displaying Bounds* on page 42, can be used to display a specified set of bounds.

Specifying Item Ranges

For some options of the `display` command, you must specify the item or range of items you want to see. Whenever input defining a range of items is required, ILOG CPLEX expects two indices separated by a hyphen (the range character `-`). The indices can be names or matrix index numbers. You simply enter the starting name (or index number), a hyphen (–), and finally the ending name (or index number). ILOG CPLEX automatically sets the default upper and lower limits defining any range to be the highest and lowest possible values. Therefore, you have the option of leaving out either the upper or lower name (or index number) on either side of the hyphen. To see every possible item, you would simply enter –.

Displaying Variable or Constraint Names

You can display a variable name by using the `display` command with the options "`problem names variables`." If you do not enter the word "`variables`," ILOG CPLEX prompts you to specify whether you wish to see a constraint or variable name.

Type:

```
display problem names variables
```

In response, ILOG CPLEX prompts you to specify a set of variable names to be displayed, like this:

```
Display which variable name(s):
```

Specify these variables by entering the names of the variables or the numbers corresponding to the columns of those variables. A single number can be used or a range such as `1-2`. All of the names can be displayed at once if you type a hyphen (the character `-`). Try this by entering a hyphen at the prompt and pressing the `<return>` key.

```
Display which variable name(s): -
```

In the example, there are three variables with default names. CPLEX displays these three names:

```
x1  x2  x3
```

If you want to see only the second and third names, you could either enter the range as `2-3` or specify everything following the second variable with `2-`. Try this technique:

```
display problem names variables
Display which variable name(s): 2-
```

```
x2  x3
```

If you enter a number without a hyphen, you will see a single variable name:

```
display problem names variables
Display which variable name(s): 2
x2
```

Summary

- You can display variable names by entering the command:

  ```
  display problem names variables
  ```

- You can display constraint names by entering the command:

  ```
  display problem names constraints
  ```

Ordering Variables

In the example problem there is a direct correlation between the variable names and their numbers (`x1` is variable 1, `x2` is variable 2, etc.); that is not always the case. The internal ordering of the variables is based on their order of occurrence when the problem is entered. For example, if `x2` had not appeared in the objective function, then the order of the variables would be `x1`, `x3`, `x2`.

You can see the internal ordering by using the hyphen when you specify the range for the `variables` option. The variables are displayed in the order corresponding to their internal ordering.

All of the options of the `display` command can be entered directly after the word `display` to eliminate intermediate steps. The following command is correct, for example:

```
display problem names variables 2-3
```

Displaying Constraints

To view a single constraint within the matrix, use the command and the constraint number. For example, type the following:

```
display problem constraints 2
```

The second constraint appears:

```
c2: x1 - 3 x2 + x3 <= 30
```

Displaying the Objective Function

When you want to display only the objective function, you must enter its name (`obj` by default) or an index number of 0.

```
display problem constraints
Display which constraint name(s): 0
```

```
Maximize
  obj: x1 + 2 x2 + 3 x3
```

Displaying Bounds

To see only the bounds for the problem, type the following command (don't forget the hyphen character):

```
display problem bounds -
```

The result is:

```
0 <= x1 <= 40
All other variables are >= 0.
```

Summary

The general syntax of the `display` command is:

```
display option [option2] identifier [identifier2]
```

Solving a Problem

The problem is now correctly entered, and ILOG CPLEX can be used to solve it. We continue our example with the following topics:

- Solving the Example Problem
- Solution Options
- Displaying Post-Solution Information

Solving the Example Problem

The `optimize` command tells ILOG CPLEX to solve the LP problem. CPLEX uses the dual simplex optimizer, unless another method has been specified by setting the `LPMETHOD` parameter.

Entering the Optimize Command

At the ILOG CPLEX prompt, type the command:

```
optimize
```

Preprocessing

First, ILOG CPLEX tries to simplify or reduce the problem using its presolver and aggregator. If any reductions are made, a message will appear. However, in our small example, no reductions are possible.

Monitoring the Iteration Log

Next, an iteration log appears on the screen. ILOG CPLEX reports its progress as it solves the problem. The solution process involves two stages:

- during Phase I, ILOG CPLEX searches for a feasible solution
- in Phase II, ILOG CPLEX searches for the optimal feasible solution.

The iteration log periodically displays the current iteration number and either the current scaled infeasibility during Phase I, or the objective function value during Phase II. Once the optimal solution has been found, the objective function value, solution time, and iteration count (total, with Phase I in parentheses) are displayed. This information can be useful for monitoring the rate of progress.

The iteration log display can be modified by the `set simplex display` command to display differing amounts of data while the problem is being solved.

Reporting the Solution

After it finds the optimal solution, ILOG CPLEX reports:

- the objective function value
- the problem solution time in seconds
- the total iteration count
- the Phase I iteration count (in parentheses)

Optimizing our example problem produces a report like the following one (although the solution times vary with each computer):

```
Tried aggregator 1 time.
No presolve or aggregator reductions.
Presolve Time = 0.00 sec.

Iteration Log . . .
Iteration:     1  Dual infeasibility =            0.000000
Iteration:     2  Dual objective     =          202.500000

Dual simplex - Optimal:  Objective =    2.0250000000e+02
Solution Time =    0.00 sec.  Iterations = 2 (1)

CPLEX>
```

In our example, ILOG CPLEX finds an optimal solution with an objective value of 202.5 in two iterations. For this simple problem, 1 Phase I iteration was required.

Summary

To solve an LP problem, use the command:

```
optimize
```

Solution Options

We describe here some of the basic options in solving linear programming problems. Although the tutorial example does not make use of these options, you will find them useful when handling larger, more realistic problems. The topics are:

- Filing Iteration Logs
- Re-Solving
- Using Alternative Optimizers
- Interrupting the Optimization Process

For detailed information on performance options, refer to the *ILOG CPLEX User's Manual*.

Filing Iteration Logs

Every time ILOG CPLEX solves a problem, much of the information appearing on the screen is also directed into a log file. This file is automatically created by ILOG CPLEX with the name `cplex.log`. If there is an existing `cplex.log` file in the directory where ILOG CPLEX is launched, ILOG CPLEX will append the current session data to the existing file. If you want to keep a unique log file of a problem session, you can change the default name with the `set logfile` command. (See the *ILOG CPLEX User's Manual.)* The log file is written in standard ASCII format and can be edited with any text editor.

Re-Solving

You may re-solve the problem by reissuing the `optimize` command. ILOG CPLEX restarts the solution process from the previous optimal basis, and thus requires zero iterations. If you do not wish to restart the problem from an advanced basis, use the `set advance` command to turn off the advanced start indicator.

Remember that a problem must be present in memory (entered via the `enter` command or read from a file) before you issue the `optimize` command.

Using Alternative Optimizers

In addition to the `optimize` command, ILOG CPLEX can use the primal simplex optimizer (`primopt` command), the dual simplex optimizer (`tranopt` command), the barrier optimizer (`baropt` command) and the network optimizer (`netopt` command). Many problems can be solved faster using these alternative optimizers, which are described in more detail in the *ILOG CPLEX User's Manual*. If you want to solve a mixed integer programming problem, the `optimize` command is equivalent to the `mipopt` command.

Interrupting the Optimization Process

Our short example was solved very quickly. However, larger problems, particularly mixed integer problems, can take much longer. Occasionally it may be useful to interrupt the optimization process. ILOG CPLEX allows such interruptions if you use `control-c`. (The `control` and `c` keys must be pressed simultaneously.) Optimization is interrupted, and

ILOG CPLEX issues a message indicating that the process was stopped and displays progress information. If you issue another optimization command in the same session, ILOG CPLEX will resume optimization from where it was interrupted.

Displaying Post-Solution Information

Once an optimal solution is found, ILOG CPLEX can provide many different kinds of information for viewing and analyzing the results. This information is accessed via the `display` command and via some `write` commands.

Information about the following is available with the `display solution` command:

- objective function value;
- solution values;
- slack values;
- reduced costs;
- dual values (shadow prices);
- basic rows and columns.

For information on the `write` commands, see *Writing Problem and Solution Files* on page 47. Sensitivity analysis can also be performed in analyzing results, as described in *Performing Sensitivity Analysis* on page 46.

For example, to view the optimal value of each variable, enter the command:

```
display solution variables –
```

In response, the list of variable names with the solution value for each variable is displayed, like this:

```
Variable Name    Solution Value
x1                     40.000000
x2                     17.500000
x3                     42.500000
```

To view the slack values of each constraint, enter the command:

```
display solution slacks -
```

The resulting message indicates that for this problem the slack variables are all zero.

```
All slacks in the range 1-2 are 0.
```

To view the dual values (sometimes called shadow prices) for each constraint, enter the command:

```
display solution dual -
```

The list of constraint names with the solution value for each constraint appears, like this:

```
Constraint Name     Dual Price
c1                    2.750000
c2                    0.250000
```

Summary

Display solution characteristics by entering a command with the syntax:

```
display solution identifier
```

Performing Sensitivity Analysis

Sensitivity analysis of the objective function and right-hand side provides meaningful insight about ways in which the optimal solution of a problem changes in response to small changes in these parts of the problem data.

Sensitivity analysis can be performed on the following:

- objective function;
- right-hand side values;
- bounds.

To view the sensitivity analysis of the objective function, enter the command:

```
display sensitivity obj -
```

For our example, ILOG CPLEX displays the following ranges for sensitivity analysis of the objective function:

```
 OBJ Sensitivity Ranges

Variable Name  Reduced Cost      Down   Current          Up
x1                   3.5000   -2.5000    1.0000   +infinity
x2                     zero   -5.0000    2.0000      3.0000
x3                     zero    2.0000    3.0000   +infinity
```

ILOG CPLEX displays each variable, its reduced cost and the range over which its objective function coefficient can vary without forcing a change in the optimal basis. The current value of each objective coefficient is also displayed for reference. Objective function sensitivity analysis is useful to determine how sensitive the optimal solution is to the cost or profit associated with each variable.

Similarly, to view sensitivity analysis of the right-hand side, type the command:

```
display sensitivity rhs -
```

For our example, ILOG CPLEX displays the following ranges for sensitivity analysis of the right-hand side (RHS):

```
 RHS Sensitivity Ranges

Constraint Name Dual Price       Down   Current          Up
c1                  2.7500   -36.6667   20.0000   +infinity
c2                  0.2500  -140.0000   30.0000    100.0000
```

ILOG CPLEX displays each constraint, its dual price, and a range over which its right-hand side coefficient can vary without changing the optimal basis. The current value of each RHS coefficient is also displayed for reference. Right-hand side sensitivity information is useful for determining how sensitive the optimal solution and resource values are to the availability of those resources.

ILOG CPLEX can also display lower bound sensitivity ranges with the command

```
display sensitivity lb
```

and upper bound sensitivity with the command

```
display sensitivity ub
```

Summary

Display sensitivity analysis characteristics by entering a command with the syntax:

```
display sensitivity identifier
```


Writing Problem and Solution Files

The problem or its solution can be saved by using the `write` command. This command writes the problem statement or a solution report to a file.

We continue with the tutorial example in the topics:

- Selecting a Write File Format
- Writing LP Files
- Writing Basis Files
- Using Path Names

Selecting a Write File Format

When you type the `write` command in the Interactive Optimizer, ILOG CPLEX displays a menu of options and prompts you for a file format, like this:

```
File Type Options:

bas          INSERT format basis file
bin          Binary solution file
dpe          Binary format for dual-perturbed problem
dua          MPS format of explicit dual of problem
emb          MPS format of (embedded) network
iis          Irreducibly inconsistent set (LP format)
lp           LP format problem file
min          DIMACS min-cost network-flow format of (embedded) network
mps          MPS format problem file
mst          MIP start file
net          CPLEX network format of (embedded) network
ord          Integer priority order file
ppe          Binary format for primal-perturbed problem
pre          Binary format for presolved problem
qp           Quadratic coefficient matrix file
rew          MPS format problem with generic names
sav          Binary matrix and basis file
sos          Special ordered sets file
tre          Branch-and-bound treesave file
txt          Text solution file
vec          Vector solution format file

File type:
```

- The BAS format is used for storing basis information and is described in *Writing Basis Files* on page 50. See also *Reading Basis Files* on page 53.
- The BIN and TXT options create solution files. The BIN option writes a binary format solution file, while the TXT option writes an ASCII text file.
- The DPE and PPE options are used for saving perturbed problems (in SAV format).
- DUA writes out the dual formulation of a problem (in MPS format).
- EMB writes a file for an embedded network (in MPS format).
- The IIS format is used for infeasibility diagnostics.
- The LP format was discussed in *Using the LP Format* on page 35. Using this format is described in *Writing LP Files* on page 49 and *Reading LP Files* on page 51.
- The MIN format was developed by DIMACS to represent minimum-cost network flow problems.
- The MPS format is described in *Reading MPS Files* on page 52.
- The MST format allows an integer feasible solution to be saved.

- NET writes a file for an embedded network (in ILOG CPLEX format).
- The ORD format allows an integer priority order file to be written.
- PRE writes out a presolved version of the problem in SAV format.
- The QP format specifies the quadratic coefficient matrix elements.
- The REW option creates an MPS format problem file with generic names.
- The SAV format is a special binary format which facilitates very fast problem reading and writing. Because SAV format is binary (rather than ASCII text), it is not possible for you to view and edit SAV files using standard editors.
- The SOS format specifies the special ordered sets of a model.
- The TRE format saves the branch-and-bound tree for restart purposes.
- The VEC format saves the solution to a pure barrier solution, as a restart to a later crossover step.

> ***Reminder:*** All these file formats are described in more detail in the *ILOG CPLEX Reference Manual*.

Writing LP Files

When you enter the `write` command. the following message appears:

```
Name of file to write:
```

Enter the problem name "example", and CPLEX will ask you to select from a list of options. For this example, choose LP. CPLEX displays a confirmation message, like this:

```
Problem written to file 'example'.
```

If you would like to save the file with a different name, you can simply use the `write` command with the new file name as an argument. Try this, using the name `example2`. This time we avoid intermediate prompts by specifying an LP problem type, like this:

```
write example2 lp
```

Another way of avoiding the prompt for a file format is by specifying the file type explicitly in the file name extension. Try the following as an example:

```
write example.lp
```

Using a file extension to indicate the file type is the recommended naming convention. This makes it easier to keep track of your problem and solution files.

When the file type is specified by the file name extension, ILOG CPLEX ignores subsequent file type information issued within the `write` command. For example, ILOG CPLEX responds to the following command by writing an LP format problem file:

```
write example.lp mps
```

Writing Basis Files

Another optional file format is BAS. Unlike the LP and MPS formats, this format is not used to store a description of the problem statement. Rather, it is used to store information about the solution to a problem, information known as a *basis*. Even after changes are made to the problem, using a prior basis to jump-start the optimization can speed solution time considerably. A basis can be written only after a problem has been solved. Try this now with the following command:

```
write example.bas
```

In response, ILOG CPLEX displays a confirmation message, like this:

```
Basis written to file 'example.bas'.
```

When a very large problem is being solved by the primal or dual simplex optimizer, a file with the format extension `.xxx` is automatically written after every 50,000 iterations (a frequency that can be adjusted by the `set simplex basisinterval` command). This periodically written basis can be useful as insurance against the possibility that a long optimization may be unexpectedly interrupted due to power failure or other causes, because the optimization can then be restarted using this advanced basis.

Using Path Names

A full path name may also be included to indicate on which drive and directory any file should be saved. The following might be a valid `write` command if the disk drive on your system contains a root directory named `problems`:

```
write /problems/example.lp
```

Summary

The general syntax for the `write` command is:

```
write filename file_format
```

or

```
write filename.file_extension
```

where *`file_extension`* indicates the format in which the file is to be saved.

Reading Problem Files

When you are using ILOG CPLEX to solve linear optimization problems, you may frequently enter problems by reading them from files instead of entering them from the keyboard.

Continuing the tutorial from *Writing Problem and Solution Files* on page 47, the topics are:

- Selecting a Read File Format
- Reading LP Files
- Using File Extensions
- Reading MPS Files
- Reading Basis Files

Selecting a Read File Format

When you type the `read` command in the Interactive Optimizer, ILOG CPLEX displays the following prompt about file formats on the screen:

```
File Type Options:

bas          INSERT format basis file
lp           LP format problem file
min          DIMACS min-cost network-flow format file
mps          MPS format problem file
mst          MIP start file
net          CPLEX Network-flow format file
ord          Integer priority order file
qp           Quadratic coefficient matrix file
sav          Binary matrix and basis file
sos          Special ordered sets file
tre          Branch-and-bound treesave file
vec          Vector solution format file

File Type Options:
```

> ***Reminder:*** All these file formats are described in more detail in the *ILOG CPLEX Reference Manual*.

Reading LP Files

At the `CPLEX>` prompt type:

```
read
```

The following message appears requesting a file name:

```
Name of file to read:
```

Four files have been saved at this point in our tutorial:

```
example
example2
example.lp
example.bas
```

Specify the file named `example` that you saved while practicing the `write` command.

You recall that the example problem was saved in LP format, so in response to the file type prompt, enter:

```
lp
```

ILOG CPLEX displays a confirmation message, like this:

```
Problem 'example' read.
Read Time =   0.03 sec.
```

The example problem is now in memory, and you can manipulate it with ILOG CPLEX commands.

> ***Tip:*** The intermediate prompts for the `read` command can be avoided by entering the entire command on one line, like this:
>
> ```
> read example lp
> ```

Using File Extensions

If the file name has an extension that corresponds to one of the supported file formats, ILOG CPLEX automatically reads it without your having to specify the format. Thus, the following command automatically reads the problem file `example.lp` in LP format:

```
read example.lp
```

Reading MPS Files

ILOG CPLEX can also read industry-standard MPS formatted files. We use a problem called `afiro.mps` (provided in the ILOG CPLEX distribution) as an example. If you include the `.mps` extension in the file name, ILOG CPLEX will recognize the file as being in MPS format. If you omit the extension, you must specify that the file is of the type MPS.

```
read afiro mps
```

Once the file has been read, the following message appears:

```
Selected objective sense:  MINIMIZE
Selected objective  name:  obj
Selected RHS        name:  rhs
Problem 'afiro' read.
Read time =    0.01 sec.
```

ILOG CPLEX reports additional information when it reads MPS formatted files. Since these files can contain multiple objective function, right-hand side, bound, and other information, ILOG CPLEX displays which of these is being used for the current problem. See the *ILOG CPLEX User's Manual* to learn more about special considerations for using MPS formatted files.

Reading Basis Files

In addition to other file formats, the `read` command is also used to read basis files. These files contain information for ILOG CPLEX that tells the simplex method where to begin the next optimization. Basis files usually correspond to the result of some previous optimization and help to speed re-optimization. They are particularly helpful when you are dealing with very large problems if small changes are made to the problem data.

Writing Basis Files on page 50 showed you how to save a basis file for the `example` after it was optimized. For this tutorial, first read the `example.lp` file. Then read this basis file by typing the following command:

```
read example.bas
```

The message of confirmation:

```
Basis 'example.bas' read.
```

indicates that the basis file was successfully read. If the advanced basis indicator is on, this basis will be used as a starting point for the next optimization, and any new basis created during the session will be used for future optimizations. If the basis changes during a session, you can save it by using the `write` command.

Summary

The general syntax for the read command is:

```
read filename file_format
```

or

```
read filename.file_extension
```

where `file_extension` corresponds to one of the allowed file formats.

Setting ILOG CPLEX Parameters

ILOG CPLEX users can vary parameters by means of the `set` command. This command is used to set ILOG CPLEX parameters to values different from their default values. The procedure for setting a parameter is similar to that of other commands. Commands can be carried out incrementally or all in one line from the ILOG CPLEX prompt. Whenever a parameter is set to a new value, ILOG CPLEX inserts a comment in the log file that indicates the new value.

Setting a Parameter

To see the parameters that can be changed, type:

```
set
```

The parameters that can be changed are displayed with a prompt, like this:

```
Available Parameters:

advance          set indicator for advanced starting information
barrier          set parameters for barrier optimization
clocktype        set type of clock used to measure time
defaults         set all parameter values to defaults
logfile          set file to which results are printed
lpmethod         set method for linear optimization
mip              set parameters for mixed integer optimization
network          set parameters for network optimizations
output           set extent and destinations of outputs
preprocessing    set parameters for preprocessing
qpmethod         set method for quadratic optimization
read             set problem read parameters
sifting          set parameters for sifting optimization
simplex          set parameters for primal and dual simplex optimizations
threads          set default parallel thread count
timelimit        set time limit in seconds
workdir          set directory for working files
workmem          set memory available for working storage (in megabytes)

Parameter to set:
```

If you press the `<return>` key without entering a parameter name, the following message is displayed:

```
No parameters changed.
```

Resetting Defaults

After making parameter changes, it is possible to reset all parameters to default values by issuing one command:

```
set defaults
```

This resets all parameters to their default values, except for the name of the log file.

Summary

The general syntax for the `set` command is:

```
set parameter option new_value
```

Displaying Parameter Settings

The current values of the parameters can be displayed with the command:

```
display settings all
```

A list of parameters with settings that differ from the default values can be displayed with the command:

```
display settings changed
```

For a description of all parameters and their default values, see the Parameter Table in the *ILOG CPLEX Reference Manual*. For examples of how to set parameters, see the *ILOG CPLEX User's Manual*.

ILOG CPLEX also accepts customized system parameter settings via a parameter specification file. See the *ILOG CPLEX User's Manual* for a description of the parameter specification file and its use.

Adding Constraints and Bounds

If you wish to add either new constraints or bounds to your problem, use the `add` command. This command is similar to the `enter` command in the way it is used, but it has one important difference: the `enter` command is used to start a brand new problem, whereas the `add` command only adds new information to the current problem.

Suppose that in the example you need to add a third constraint:

$$x_1 + 2x_2 + 3x_3 \geq 50$$

You may do either interactively or from a file.

Adding Interactively

Type the `add` command, then enter the new constraint on the blank line. After validating the constraint, the cursor moves to the next line. You are in an environment identical to that of the `enter` command after having issued `subject to`. At this point you may continue to add constraints or you may type `bounds` and enter new bounds for the problem. For the present example, type `end` to exit the `add` command. Your session should look like this:

```
add
Enter new constraints and bounds ['end' terminates]:
x1 + 2x2 + 3x3 >= 50
end
Problem addition successful.
```

When the problem is displayed again, the new constraint appears, like this:

```
display problem all

Maximize
 obj: x1 + 2 x2 + 3 x3
Subject To
 c1: - x1 +   x2 +   x3 <= 20
 c2: x1 - 3 x2 +   x3 <= 30
 c3: x1 + 2 x2 + 3 x3 >= 50
Bounds
 0 <= x1 <= 40
 All other variables are >= 0.
end
```

Adding from a File

Alternatively, you may read in new constraints and bounds from a file. If you enter a file name after the `add` command, ILOG CPLEX will read a file matching that name. The file contents must comply with standard ILOG CPLEX LP format. ILOG CPLEX does not prompt for a file name if none is entered. Without a file name, interactive entry is assumed.

Summary

The general syntax for the `add` command is:

```
   add
```
or
```
   add filename
```

Changing a Problem

The `enter` and `add` commands allow you to build a problem from the keyboard, but they do not allow you to change what you have built. You make changes with the `change` command.

The `change` command can be used for:

- Changing Constraint or Variable Names
- Changing Sense
- Changing Bounds and Removing Bounds
- Changing Coefficients
- Deleting entire constraints or variables

Start out by changing the name of the constraint that you added with the `add` command. In order to see a list of change options, type:

```
change
```

The elements that can be changed are displayed like this:

```
Change options:

bounds           change bounds on a variable
coefficient      change a coefficient
delete           delete some part of the problem
name             change a constraint or variable name
objective        change objective function value
problem          change problem type
qpterm           change a quadratic objective term
rhs              change a right-hand side or network supply/demand value
sense            change objective function or a constraint sense
type             change variable type

Change to make:
```


Changing Constraint or Variable Names

Enter `name` at the `Change to make:` prompt to change the name of a constraint:

```
Change to make: name
```

The present name of the constraint is `c3`. In the example, you can change the name to `new3` to differentiate it from the other constraints using the following entries:

```
Change a constraint or variable name ['c' or 'v']: c
Present name of constraint: c3
New name of constraint: new3
The constraint 'c3' now has name 'new3'.
```

The name of the constraint has been changed.

The problem can be checked with a `display` command (for example, `display problem constraints new3`) to confirm that the change was made.

This same technique can also be used to change the name of a variable.

Changing Sense

Next, change the sense of the `new3` constraint from ≥ to ≤using the `sense` option of the `change` command. At the `CPLEX>` prompt type:

```
change sense
```

ILOG CPLEX prompts you to specify a constraint. There are two ways of specifying this constraint: if you know the name (for example, `new3`), you can enter the name; if you do not know the name, you can specify the number of the constraint. In this example, the number is 3 for the `new3` constraint. Try the first method and type:

```
Change sense of which constraint: new3
Sense of constraint 'new3' is '>='.
```

ILOG CPLEX tells you the current sense of the selected constraint. All that is left now is to enter the new sense, which can be entered as <=, >=, or =. You can also type simply < (interpreted as $\leq$) or > (interpreted as $\geq$). The letters `l`, `g`, and `e` are also interpreted as $\leq$, $\geq$, and = respectively.

```
New sense ['<=' or '>=' or '=']: <=
Sense of constraint 'new3' changed to '<='.
```

The sense of the constraint has been changed.

The sense of the objective function may be changed by specifying the objective function name (its default is `obj`) or the number 0 when ILOG CPLEX prompts you for the constraint. You are then prompted for a new sense. The sense of an objective function can take the value `maximum` or `minimum` or the abbreviation `max` or `min`.

Changing Bounds

When the example was entered, bounds were set specifically only for the variable `x1`. The bounds can be changed on this or other variables with the `bounds` option. Again, start by selecting the command and option.

```
change bounds
```

Select the variable by name or number and then select which bound you would like to change. For the example, change the upper bound of variable x_2 from $+\infty$to 50.

```
Change bounds on which variable: x2
Present bounds on variable x2: The indicated variable is >= 0.
Change lower or upper bound, or both ['l', 'u', or 'b']: u
Change upper bound to what ['+inf' for no upper bound]: 50
New bounds on variable 'x2':  0 <= x2 <= 50
```

Removing Bounds

To remove a bound, set it to $+\infty$or $-\infty$. Interactively, use the identifiers `inf` and `-inf` instead of the symbols. To change the upper bound on `x2` back to $+\infty$, use the one line command:

```
change bounds x2 u inf
```

You receive the message:

```
New bounds on variable 'x2': The indicated variable is >= 0.
```

The bound is now the same as it was when the problem was originally entered.

Changing Coefficients

Up to this point all of the changes that have been made could be referenced by specifying a single constraint or variable. In changing a coefficient, however, a constraint *and* a variable

must be specified in order to identify the correct coefficient. As an example, let's change the coefficient of x3 in the new3 constraint from 3 to 30.

As usual, you must first specify which change command option to use:

```
change coefficient
```

You must now specify both the constraint row and the variable column identifying the coefficient you wish to change. Enter both the constraint name (or number) and variable name (or number) on the same line, separated by at least one space. The constraint name is new3 and the variable is number 3, so in response to the following prompt, type new3 and 3, like this, to identify the one to change:

```
Change which coefficient ['constraint' 'variable']: new3 3
Present coefficient of constraint 'new3', variable '3' is 3.000000.
```

The final step is to enter the new value for the coefficient of x3.

```
Change coefficient of constraint 'new3', variable '3' to what: 30
Coefficient of constraint 'new3', variable '3' changed to 30.000000.
```


Objective & RHS Coefficients

To change a coefficient in the objective function, or in the right-hand side, use the corresponding change command option, objective or rhs. For example, to specify the right-hand side of constraint 1 to be 25.0, we could enter the following (but for this tutorial, do not enter this now):

```
change rhs 1 25.0
```

Deleting

Another option to the change command is delete. This option is used to remove an entire constraint or a variable from a problem. Return the problem to its original form by removing the constraint you added earlier. Type:

```
change delete
```

ILOG CPLEX displays a list of delete options.

```
Delete Options:

constraints     delete range of constraints
variables       delete range of variables
equality        delete range of equality constraints
greater-than    delete range of greater-than constraints
less-than       delete range of less-than constraints
```

At the first prompt, specify that you want to delete a constraint.

```
Deletion to make: constraints
```

At the next prompt, enter a constraint name or number, or a range as you did when you used the `display` command. Since the constraint we want to delete is named `new3`, we enter that name:

```
Delete which constraint(s): new3
Constraint 3 deleted.
```

Check to be sure that the correct range or number is specified when you perform this operation, since constraints are permanently removed from the problem. Indices of any constraints that appeared after a deleted constraint will be decremented to reflect the removal of that constraint.

The last message indicates that the operation is complete. The problem can now be checked to see if it has been changed back to its original form.

```
display problem all

Maximize
 obj:  x1 + 2 x2 + 3 x3
Subject To
 c1: - x1 +   x2 +   x3 <= 20
 c2:   x1 - 3 x2 +   x3 <= 30
Bounds
 0 <= x1 <= 40
All other variables are >= 0.
```

When you remove a constraint with the `delete` option, that constraint no longer exists in memory; however, variables that appear in the deleted constraint are not removed from memory. If a variable from the deleted constraint appears in the objective function, it may still influence the solution process. If that is not what you want, these variables can be explicitly removed using the `delete` option.

Summary

The general syntax for the `change` command is:

```
change option identifier [identifier2] new value
```

Executing Operating System Commands

The execute command (“`xecute`”) is simple but useful. It is used to execute operating system commands outside of the ILOG CPLEX environment. By using `xecute`, you avoid having to save a problem and quit ILOG CPLEX in order to carry out a system function (such as viewing a directory, for example).

As an example, if you wanted to check whether all of the files saved in the last session are really in the current working directory, the following ILOG CPLEX command shows the contents of the current directory in a UNIX operating system, using the UNIX command `ls`:

```
xecute ls -l
total 7448
-r--r--r--    1      3258 Jul 14 10:34 afiro.mps
-rwxr-xr-x    1   3783416 Apr 22 10:32 cplex
-rw-r--r--    1      3225 Jul 14 14:21 cplex.log
-rw-r--r--    1       145 Jul 14 11:32 example
-rw-r--r--    1       112 Jul 14 11:32 example.bas
-rw-r--r--    1       148 Jul 14 11:32 example.lp
-rw-r--r--    1       146 Jul 14 11:32 example2
```

After the command is executed, the `CPLEX>` prompt returns, indicating that you are still in ILOG CPLEX. Most commands that can normally be entered from the prompt for your operating system can also be entered with the `xecute` command. The command may be as simple as listing the contents of a directory or printing the contents of a file, or as complex as starting a text editor to modify a file. Anything that can be entered on one line after the operating system prompt can also be executed from within ILOG CPLEX. However, this command differs from other ILOG CPLEX commands in that it must be entered on a single line. No prompt will be issued. In addition, the operating system may fail to carry out the command if insufficient memory is available. In that case, no message is issued by the operating system, and the result is a return to the `CPLEX>` prompt.

Summary

The general syntax for the `xecute` command is:

```
xecute command line
```

Quitting ILOG CPLEX

When you are finished using ILOG CPLEX and want to leave it, type:

```
quit
```

If a problem has been modified, be sure to save the file before issuing a `quit` command. ILOG CPLEX will not prompt you to save your problem.

C H A P T E R

3

Concert Technology Tutorial for C++ Users

This tutorial shows you how to write C++ programs using CPLEX with Concert Technology. In this chapter you will learn about:

- The Design of CPLEX in Concert Technology
- Compiling and Linking CPLEX in Concert Technology Applications
- The Anatomy of a Concert Technology Application
- Building and Solving a Small LP Model in C++
- Writing and Reading Models and Files
- Selecting an Optimizer
- Reading a Problem from a File: Example ilolpex2.cpp
- Modifying and Reoptimizing
- Modifying an Optimization Problem: Example ilolpex3.cpp

The Design of CPLEX in Concert Technology

A clear understanding of C++ **objects** is fundamental to using Concert Technology with CPLEX to build and solve optimization models. These objects can be divided into two categories:

1. **Modeling objects** are used to define the optimization problem. Generally an application creates multiple modeling objects to specify one optimization problem. Those objects are grouped into an **`IloModel` object** representing the complete optimization problem.
2. **`IloCplex` objects** are used to solve the problems that have been created with the modeling objects. An `IloCplex` object reads a model and extracts its data to the appropriate representation for the CPLEX optimizer. Then the `IloCplex` object is ready to solve the model it extracted and be queried for solution information.

Thus, the modeling and optimization parts of a user-written application program are represented by a group of interacting C++ objects created and controlled within the application. Figure 3.1 shows a picture of an application using CPLEX with Concert Technology to solve optimization problems.

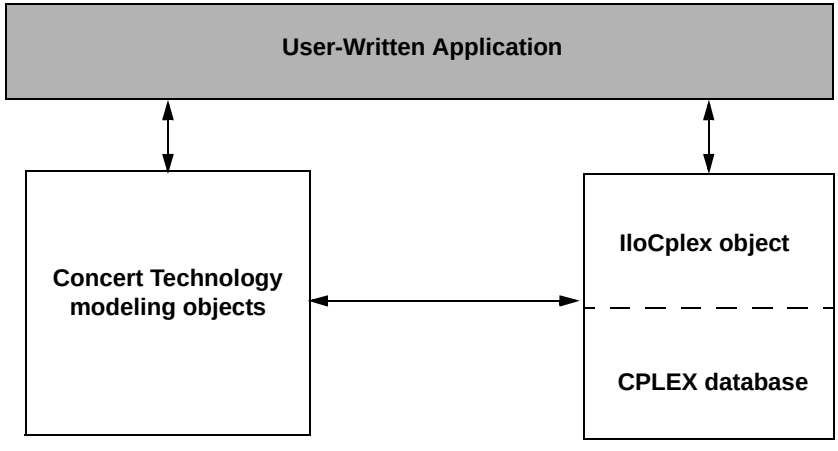

Figure 3.1 *A View of CPLEX with Concert Technology*

The CPLEX database includes the computing environment, its communication channels, and your problem objects.

In this chapter we give a brief tutorial illustrating the modeling and solution classes provided by Concert Technology and CPLEX. More extensive coverage of the modeling classes can be found in the chapter about `IloCplex` in the *ILOG Concert Technology User's Manual* and in the *ILOG Concert Technology Reference Manual*. More information about the

algorithm class `IloCplex` and its nested classes can be found in the *ILOG CPLEX User's Manual* and *ILOG CPLEX Reference Manual*.

Compiling and Linking CPLEX in Concert Technology Applications

To exploit a C++ library like ILOG CPLEX in Concert Technology, you need to tell your *compiler* where to find the ILOG CPLEX and Concert include files (that is, the header files), and you also need to tell the *linker* where to find the ILOG CPLEX and Concert libraries. The sample projects and `makefiles` illustrate how to carry out these crucial steps for the examples in the standard distribution. They use relative path names to indicate to the compiler where the header files are, and to the linker where the libraries are.

Testing Your Installation on UNIX

To run the test, follow these steps.

1. First check the readme.html file in the standard distribution to locate the right subdirectory containing a `makefile` appropriate for your platform.
2. Go to that subdirectory.
3. Then use the sample `makefile` located there to compile and link the examples that came in the standard distribution.
4. Execute one of the compiled examples.

Testing Your Installation on Windows

To run the test on a Windows platform, first consult the `readme.html` file in the standard distribution. That file will tell you where to find another text file that contains information about your particular platform. That second file will have an abbreviated name that corresponds to a particular combination of machine, architecture, and compiler. For example, if you are working on a personal computer with Windows NT and Microsoft Visual C++ compiler, version 6, then the `readme.html` file will direct you to the `msvc.html` file where you will find detailed instructions about how to create a project to compile, link, and execute the examples in the standard distribution.

The examples have been tested repeatedly on all the platforms compatible with ILOG CPLEX, so if you successfully compile, link, and execute them, then you can be sure that your installation is correct.

In Case of Problems

If you encounter difficulty when you try this test, then there is a problem in your installation, and you need to correct it before you begin real work with ILOG CPLEX.

For example, if you get a message from the compiler such as

```
ilolpex3.cpp 1: Can't find include file ilcplex/ilocplex.h
```

then you need to verify that your compiler knows where you have installed ILOG CPLEX and its include files (that is, its header files).

If you get a message from the linker, such as

```
ld: -lcplex: No such file or directory
```

then you need to verify that your linker knows where the ILOG CPLEX library is located on your system.

If you get a message such as

```
ilm: CPLEX: no license found for this product
```

or

```
ilm: CPLEX: invalid encrypted key "MNJVUXTDJV82" in "/usr/ilog/ilm/
access.ilm";run ilmcheck
```

then there is a problem with your license to use ILOG CPLEX. Review the *ILOG License Manager User's Guide and Reference* to see whether you can correct the problem. If not, call the technical support hotline and repeat the error message there.

If you successfully compile, link, and execute one of the examples in the standard distribution, then you can be sure that your installation is correct, and you can begin to use ILOG CPLEX in Concert Technology seriously.

The Anatomy of a Concert Technology Application

Concert Technology is a C++ class library, and therefore Concert Technology applications consist of interacting C++ objects. This section gives a short introduction to the most important classes that are usually found in a complete Concert Technology CPLEX application.

Constructing the Environment — IloEnv

An `IloEnv` is typically the first object created in any Concert Technology application.

You construct an IloEnv object by declaring a variable of type IloEnv. For example, to create an environment named env, you do this:

```
IloEnv env;.
```

> ***Note:*** *The environment object created in a Concert Technology application is different from the environment created in the CPLEX C library by calling the routine* CPXopenCPLEX().

The environment object is of central importance and needs to be available to the constructor of all other Concert Technology classes because (among other things) it provides optimized memory management for objects of Concert Technology classes. This provides a boost in performance compared to using the system memory management system.

As is the case for most Concert Technology classes, IloEnv is a *handle class*. This means that the variable env is a pointer to an implementation object, which is created at the same time as env in the above declaration. One advantage of using handles is that if you assign handle objects, all that is assigned is a pointer. So the statement

```
IloEnv env2 = env;
```

creates a second handle pointing to the implementation object that env already points to. Hence there may be an arbitrary number of IloEnv handle objects all pointing to the same implementation object. When terminating the Concert Technology application, the implementation object must be destroyed as well. This must be done explicitly by the user by calling

```
env.end();
```

for just *ONE* of the IloEnv handles pointing to the implementation object to be destroyed. The call to env.end() is generally the last Concert Technology operation in an application.

Creating a Model — IloModel

After creating the environment, we are ready to create one or more optimization models. Doing so consists of creating a set of modeling objects to describe each optimization model.

Modeling objects, like IloEnv objects, are handles to implementation objects. Though you will be dealing only with the handle objects, it is the implementation objects that contain the data that specifies the optimization model. If you need to remove an implementation object from memory, you need to call the end() method for one of its handle objects.

We also refer to modeling objects as *extractables*. This is because it is the individual modeling objects that are extracted one by one when you extract an optimization model to IloCplex. So, extractables are characterized by the possibility of being extracted to algorithms such as IloCplex. In fact, they all are inherited from the class IloExtractable. In other words, IloExtractable is the base class of all classes of extractables or modeling objects.

The most fundamental extractable class is `IloModel`. Objects of this class are used to describe a complete optimization model that can later be extracted to an `IloCplex` object. You create a model by constructing a variable of type `IloModel`. For example, to construct a modeling object named `model`, within an existing environment named `env`, you would do the following:

```
IloModel model(env);
```

At this point we would like to point out that the environment is passed as a parameter to the constructor. There is also a constructor that does not use the environment parameter, but this constructor creates an empty handle, the handle corresponding to a NULL pointer. Empty handles cannot be used for anything but for assigning other handles to them. We mention this, because it is a common mistake to try to use empty handles for other things.

Once an `IloModel` object has been constructed, it is populated with the extractables that define the optimization model. The most important classes here are:

`IloNumVar`	representing modeling variables;
`IloRange`	describing constraints of the form *l* <= *expr* <= *u*, where *expr* is some sort of linear expression; and
`IloObjective`	representing an objective function.

You create objects of these classes for each variable, constraint, and objective function of your optimization problem. Then you add the objects to the model by calling

```
model.add(obj);
```

for each extractable `obj`. There is no need to explicitly add the variable objects to a model, as they are implicitly considered when they are used in the range constraints (instances of `IloRange`) or the objective. At most one objective can be used in a model with `IloCplex`.

Modeling variables are constructed as objects of class `IloNumVar`, by defining variables of type `IloNumVar`. Concert Technology provides several constructors for doing this; the most flexible version, for example, is:

```
IloNumVar x1(env, 0.0, 40.0, ILOFLOAT);
```

This definition creates the modeling variable `x1` with lower bound 0.0, upper bound 40.0 and type `ILOFLOAT`, which indicates the variable is continuous. Other possible variable types include `ILOINT` for integer variables and `ILOBOOL` for boolean variables.

For each variable in the optimization model a corresponding object of class `IloNumVar` must be created. Concert Technology provides a wealth of ways to help you construct all the `IloNumVar` objects.

Once all the modeling variables have been constructed, they can be used to build expressions, which in turn are used to define objects of class `IloObjective` and `IloRange`. For example,

```
IloObjective obj = IloMinimize(env, x1 + 2*x2 + 3*x3);
```

This creates the extractable `obj` of type `IloObjective` which represents the objective function of the example presented in *Introducing ILOG CPLEX*.

Let us look in more detail what this line does. The function `IloMinimize` takes the environment and an expression as arguments, and constructs a new `IloObjective` object from it that describes the objective function to minimize the expression. This new object is returned and assigned to the new handle `obj`.

After an objective extractable is created, it must be added to the model. As noted above this is done with the `add()` method of `IloModel`. If this is all we need variable `obj` for, we can instead write more compactly:

```
model.add(IloMinimize(env, x1 + 2*x2 + 3*x3));
```

This way there is no need for the program variable `obj` and the program is shorter. If in contrast, the objective function is needed later, for example, to change it and reoptimize the model when doing scenario analysis, the variable `obj` must be created in order to refer to the objective function. (From the standpoint of algorithmic efficiency, the two approaches are comparable.)

Creating constraints and adding them to the model can be done just as easily with the following statement:

```
model.add(-x1 + x2 + x3 <= 20);
```

The part `-x1 + x2 + x3 <= 20` creates an object of class `IloRange` that is immediately added to the model by passing it to the method `IloModel::add()`. Again, if a reference to the `IloRange` object is needed later, an `IloRange` handle object must be stored for it. Concert Technology provides flexible array classes for storing data, such as these `IloRange` objects. As with variables, Concert Technology provides a variety of constructors that help create range constraints.

While the above examples use expressions with modeling variables directly for modeling, it should be pointed out that such expressions are themselves represented by yet another Concert Technology class, `IloExpr`. Like most Concert Technology objects, `IloExpr` objects are handles. Consequently, method `end()` must be called when the object is no longer needed. The only exceptions are implicit expressions, where the user does not create an `IloExpr` object, such as when writing (for example) `x1 + 2*x2`. For such implicit expressions, method `end()` should not be called. The importance of the class `IloExpr` becomes clear when expressions can no longer be fully spelled out in the source code but need instead to be built up in a loop. Operators like `+=` provide an efficient way to do this.

Solving the Model — IloCplex

Once the optimization problem has been created in an `IloModel` object, it is time to create the `IloCplex` object for solving the problem. This is done by creating a variable of type `IloCplex`. For example, to create an object named `cplex`, do the following:

```
IloCplex cplex(env);
```

again using the environment `env` as parameter. The CPLEX object can then be used to extract the model to be solved. This can be done by calling `cplex.extract(model)`. However, we recommend a shortcut that performs the construction of the `cplex` object and the extraction of the model in one line:

```
IloCplex cplex(model);
```

This works because the modeling object `model` contains within it the reference to the environment named `env`.

After this line, object `cplex` is ready to solve the optimization problem described by `model`. Solving the model is done by calling:

```
cplex.solve();
```

This method returns an `IloBool` value, where `IloTrue` indicates that `cplex` successfully found a feasible (yet not necessarily optimal) solution, and `IloFalse` indicates that no solution was found. More precise information about the outcome of the last call to method `solve()` can be obtained by calling:

```
cplex.getStatus();
```

The returned value tells you what CPLEX found out about the model: whether it found the optimal solution or only a feasible solution, whether it proved the model to be unbounded or infeasible, or whether nothing at all has been determined at this point. Even more detailed information about the termination of the solve call is available through method `IloCplex::getCplexStatus()`.

Querying Results

After successfully solving the optimization problem, you probably are interested in accessing the solution. The following methods can be used to query the solution value for a variable or a set of variables:

```
IloNum IloCplex::getValue(IloNumVar var) const;
void IloCplex::getValues(IloNumArray val,
                         const IloNumVarArray var) const;
```

For example:

```
IloNum val1 = cplex.getValue(x1);
```

stores the solution value for the modeling variable `x1` in variable `val1`. Other methods are available for querying other solution information. For example, the objective function value of the solution can be accessed using:

```
IloNum objval = cplex.getObjValue();
```

Handling Errors

Concert Technology provides two lines of defense for dealing with error conditions, suited for addressing two kind of errors. The first kind covers simple programming errors. Examples of this kind are: trying to use empty handle objects or passing arrays of incompatible lengths to functions.

This kind of error is usually an oversight and should not occur in a correct program. In order not to pay any runtime cost for correct programs asserting such conditions, the conditions are checked using `assert()` statements. The checking is disabled for production runs if compiled with the `-DNDEBUG` compiler option.

The second kind of error is more complex and cannot generally be avoided by correct programming. An example is memory exhaustion. The data may simply require too much memory, even when the program is correct. This kind of error is always checked at runtime. In cases where such an error occurs, Concert Technology throws a C++ exception.

In fact, Concert Technology provides a hierarchy of exception classes that all derive from the common base class `IloException`. Exceptions derived from this class are the only kind of exceptions that are thrown by Concert Technology. The exceptions thrown by `IloCplex` objects all derive from class `IloAlgorithm::Exception` or `IloCplex::Exception`.

To gracefully handle exceptions in a Concert Technology application we advise including all of the code in a `try/catch` clause:

```
IloEnv env;
try {
// ...
} catch (IloException& e) {
cerr << "Concert Exception: " << e << endl;
} catch (...) {
cerr <<  "Other Exception" << endl;
}
env.end();
```

> ***Note:*** *The construction of the environment comes before the* `try/catch` *clause. In case of an exception,* `env.end()` *must still be called. To protect against failure during the construction of the environment, another* `try/catch` *clause may be added.*

If code other than Concert Technology code is used in the part of the above example denoted by `...`, we catch all other exceptions with the statement `catch(...)`. Doing so is good practice, as it assures that no exception is unhandled.

Building and Solving a Small LP Model in C++

A complete example of building and solving a small LP model can now be presented. This example demonstrates:

- General Structure of a CPLEX Concert Technology Application
- Modeling by Rows
- Modeling by Columns
- Modeling by Nonzero Elements

Example `ilolpex1.cpp`, which is one of the example programs in the standard CPLEX distribution, is an extension of the example presented in *Introducing ILOG CPLEX*. It shows three different ways of creating a Concert Technology LP model, how to solve it using `IloCplex`, and how to access the solution. Here is the problem that the example optimizes:

Maximize	$x_1 + 2x_2 + 3x_3$
subject to	$-x_1 + x_2 + x_3 \leq 20$ $x_1 - 3x_2 + x_3 \leq 30$
with these bounds	$0 \leq x_1 \leq 40$ $0 \leq x_2 \leq +\infty$ $0 \leq x_3 \leq +\infty$

General Structure of a CPLEX Concert Technology Application

The first operation is to create the environment object `env`, and the last operation is to destroy it by calling `env.end()`. The rest of the code is enclosed in a `try/catch` clause to gracefully handle any errors that may occur.

First the example creates the model object and, after checking the correctness of command line parameters, it creates empty arrays for storing the variables and range constraints of the optimization model. Then, depending on the command line parameter, the example calls one of the functions `populatebyrow()`, `populatebycolumn()`, or `populatebynonzero()`, to fill the model object with a representation of the optimization problem. These functions return the variable and range objects in the arrays `var` and `con` which are passed to them as parameters.

After the model has been populated, the `IloCplex` algorithm object `cplex` is created and the model is extracted to it. The following call of method `solve()` invokes the optimizer. If it fails to generate a solution, an error message is issued to the error stream of the environment, `cplex.error()`, and the integer -1 thrown as exception.

`IloCplex` provides the output streams `out()` for general logging, `warning()` for warning messages, and `error()` for error messages. They are preconfigured to `cout`, `cerr`, and

cerr respectively. Thus by default you will see logging output on the screen when invoking the method solve(). This can be turned off by calling cplex.setOut(env.getNullStream()), that is, by redirecting the out() stream of the IloCplex object cplex to the null stream of the environment.

If a solution is found, solution information is output through the channel, env.out() which is initialized to cout by default. The output operator << is defined for type IloAlgorithm::Status as returned by the call to cplex.getStatus(). It is also defined for IloNumArray, the Concert Technology class for an array of numerical values, as returned by the calls to cplex.getValues(), cplex.getDuals(), cplex.getSlacks(), and cplex.getReducedCosts(). In general, the output operator is defined for any Concert Technology array of elements if the output operator is defined for the elements.

The functions named populateby* are purely about modeling and are completely decoupled from the algorithm IloCplex. In fact, they don't use the cplex object, which is created only after executing one of these functions.

Modeling by Rows

The function populatebyrow creates the variables and adds them to the array x. Then the objective function and the constraints are created using expressions on the variables stored in x. The range constraints are also added to the array of constraints c. The objective object and the constraints are added to the model.

Modeling by Columns

Function populatebycolumn can be viewed as the transpose of populatebyrow. While for simple examples like this one population by rows may seem the most straightforward and natural approach, there are some models where modeling by column is a more natural or more efficient approach.

When modeling by columns, range objects are created with their lower and upper bound only. No expression is given—which is impossible since the variables are not yet created. Similarly, the objective function is created with only its intended optimization sense, and without any expression. Next the variables are created and installed in the already existing ranges and objective.

The description of how the newly created variables are to be installed in the ranges and objective is by means of *column expressions*, which are represented by the class IloNumColumn. Column expressions consist of objects of class IloAddNumVar linked together with operator +. These IloAddNumVar objects are created using operator() of the classes IloObjective and IloRange. They describe how to install a new variable to the invoking objective or range objects. For example obj(1.0) creates an IloAddNumVar

capable of adding a new modeling variable with a linear coefficient of 1.0 to the expression in `obj`. Column expressions can be built in loops using operator `+=`.

Column expressions (objects of class `IloNumColumn`) are handle objects, like most other Concert Technology objects. The method `end()` must therefore be called to delete the associated implementation object when it is no longer needed. However, for implicit column expressions, where no `IloNumColumn` object is explicitly created, such as the ones used in this example, method `end()` should not be called.

The column expression is passed as a parameter to the constructor of class `IloNumVar`. For example the constructor `IloNumVar(obj(1.0) + c[0](-1.0) + c[1]( 1.0), 0.0, 40.0)` creates a new modeling variable with lower bound 0.0, upper bound 40.0 and, by default, type `ILOFLOAT`, and adds it to the objective `obj` with a linear coefficient of 1.0, to the range `c[0]` with a linear coefficient of -1.0 and to `c[1]` with a linear coefficient of 1.0. Column expressions can be used directly to construct numerical variables with default bounds `[0, IloInfinity]` and type `ILOFLOAT`, as in the following statement:

```
x.add(obj(2.0) + c[0]( 1.0) + c[1](-3.0));
```

where `IloNumVar` does not need to be explicitly written. Here, the C++ compiler recognizes that an `IloNumVar` object needs to be passed to the `add` method and therefore automatically calls the constructor `IloNumVar(IloNumColumn)` in order to create the variable from the column expression.

Modeling by Nonzero Elements

The last of the three functions that can be used to build the model is `populatebynonzero()`. It creates objects for the objective and the ranges without expressions, and variables without columns. Then methods `IloObjective::setCoef()` and `IloRange::setCoef()` are used to set individual nonzero values in the expression of the objective and the range constraints. As usual, the objective and ranges must be added to the model.

Complete Program

The complete program follows. You can also view it online in the file `ilolpex1.cpp`.

```
// ------------------------------------------------------------- -*- C++ -*-
// File: examples/src/ilolpex1.cpp
// Version 8.1
// ------------------------------------------------------------------------
//  Copyright (C) 1999-2002 by ILOG.
//  All Rights Reserved.
//  Permission is expressly granted to use this example in the
//  course of developing applications that use ILOG products.
// ------------------------------------------------------------------------
//
// ilolpex1.cpp - Entering and optimizing a problem.  Demonstrates different
```

```
// methods for creating a problem.  The user has to choose the method
// on the command line:
//
//    ilolpex1  -r     generates the problem by adding rows
//    ilolpex1  -c     generates the problem by adding columns
//    ilolpex1  -n     generates the problem by adding a list of coefficients

#include <ilcplex/ilocplex.h>
ILOSTLBEGIN

static void
   usage (const char *progname),
   populatebyrow     (IloModel model, IloNumVarArray var, IloRangeArray con),
   populatebycolumn  (IloModel model, IloNumVarArray var, IloRangeArray con),
   populatebynonzero (IloModel model, IloNumVarArray var, IloRangeArray con);

int
main (int argc, char **argv)
{
   IloEnv   env;
   try {
      IloModel model(env);

      if (( argc != 2 )                          ||
          ( argv[1][0] != '-' )                  ||
          ( strchr ("rcn", argv[1][1]) == NULL )   ) {
         usage (argv[0]);
         throw(-1);
      }

      IloNumVarArray var(env);
      IloRangeArray con(env);

      switch (argv[1][1]) {
         case 'r':
            populatebyrow (model, var, con);
            break;
         case 'c':
            populatebycolumn (model, var, con);
            break;
         case 'n':
            populatebynonzero (model, var, con);
            break;
      }

      IloCplex cplex(model);

      // Optimize the problem and obtain solution.
      if ( !cplex.solve() ) {
         env.error() << "Failed to optimize LP" << endl;
         throw(-1);
      }
```

```
      IloNumArray vals(env);
      env.out() << "Solution status = " << cplex.getStatus() << endl;
      env.out() << "Solution value  = " << cplex.getObjValue() << endl;
      cplex.getValues(vals, var);
      env.out() << "Values        = " << vals << endl;
      cplex.getSlacks(vals, con);
      env.out() << "Slacks        = " << vals << endl;
      cplex.getDuals(vals, con);
      env.out() << "Duals         = " << vals << endl;
      cplex.getReducedCosts(vals, var);
      env.out() << "Reduced Costs = " << vals << endl;

      cplex.exportModel("lpex1.lp");
   }
   catch (IloException& e) {
      cerr << "Concert exception caught: " << e << endl;
   }
   catch (...) {
      cerr << "Unknown exception caught" << endl;
   }

   env.end();

   return 0;
}  // END main


static void usage (const char *progname)
{
   cerr << "Usage: " << progname << " -X" << endl;
   cerr << "   where X is one of the following options:" << endl;
   cerr << "      r          generate problem by row" << endl;
   cerr << "      c          generate problem by column" << endl;
   cerr << "      n          generate problem by nonzero" << endl;
   cerr << " Exiting..." << endl;
} // END usage


// To populate by row, we first create the variables, and then use them to
// create the range constraints and objective.

static void
populatebyrow (IloModel model, IloNumVarArray x, IloRangeArray c)
{
   IloEnv env = model.getEnv();

   x.add(IloNumVar(env, 0.0, 40.0));
   x.add(IloNumVar(env));
   x.add(IloNumVar(env));
   model.add(IloMaximize(env, x[0] + 2 * x[1] + 3 * x[2]));

   c.add( - x[0] +     x[1] + x[2] <= 20);
```

```
   c.add(   x[0] - 3 * x[1] + x[2] <= 30);
   model.add(c);

}  // END populatebyrow


// To populate by column, we first create the range constraints and the
// objective, and then create the variables and add them to the ranges and
// objective using column expressions.

static void
populatebycolumn (IloModel model, IloNumVarArray x, IloRangeArray c)
{
   IloEnv env = model.getEnv();

   IloObjective obj = IloMaximize(env);
   c.add(IloRange(env, -IloInfinity, 20.0));
   c.add(IloRange(env, -IloInfinity, 30.0));

   x.add(IloNumVar(obj(1.0) + c[0](-1.0) + c[1]( 1.0), 0.0, 40.0));
   x.add(IloNumVar(obj(2.0) + c[0]( 1.0) + c[1](-3.0)));
   x.add(IloNumVar(obj(3.0) + c[0]( 1.0) + c[1]( 1.0)));

   model.add(obj);
   model.add(c);

}  // END populatebycolumn


// To populate by nonzero, we first create the rows, then create the
// columns, and then change the nonzeros of the matrix 1 at a time.

static void
populatebynonzero (IloModel model, IloNumVarArray x, IloRangeArray c)
{
   IloEnv env = model.getEnv();

   IloObjective obj = IloMaximize(env);
   c.add(IloRange(env, -IloInfinity, 20.0));
   c.add(IloRange(env, -IloInfinity, 30.0));

   x.add(IloNumVar(env, 0.0, 40.0));
   x.add(IloNumVar(env));
   x.add(IloNumVar(env));

   obj.setCoef(x[0], 1.0);
   obj.setCoef(x[1], 2.0);
   obj.setCoef(x[2], 3.0);

   c[0].setCoef(x[0], -1.0);
   c[0].setCoef(x[1],  1.0);
   c[0].setCoef(x[2],  1.0);
```

```
      c[1].setCoef(x[0],  1.0);
      c[1].setCoef(x[1], -3.0);
      c[1].setCoef(x[2],  1.0);

      model.add(obj);
      model.add(c);

   }  // END populatebynonzero
```

Writing and Reading Models and Files

In example `ilolpex1.cpp` we left one line unexplained:

```
cplex.exportModel("lpex1.lp");
```

This statement causes `cplex` to write the model it has currently extracted to the file called `lpex1.lp`. In this case, the file will be written in LP format (whose usage is described in detail in the *ILOG CPLEX Reference Manual*). Other formats supported for writing problems to a file are MPS and SAV (also described in the *ILOG CPLEX Reference Manual*). `IloCplex` decides which file format to write based on the extension of the file name.

`IloCplex` also supports reading of files through one of its `importModel` methods. A call to `cplex.importModel(model, "file.lp")` causes CPLEX to read a problem from the file `file.lp` and add all the data in it to `model` as new objects. (Again, MPS and SAV format files are also supported.) In particular, CPLEX creates an instance of

`IloObjective`	for the objective function found in `file.lp`,
`IloNumVar`	for each variable found in `file.lp`, except
`IloSemiContVar`	for each semi-continuous or semi-integer variable found in `file.lp`,
`IloRange`	for each row found in `file.lp`,
`IloSOS1`	for each SOS of type 1 found in `file.lp`, and
`IloSOS2`	for each SOS of type 2 found in `file.lp`.

If you also need access to the modeling objects created by `importModel()`, two additional signatures are provided:

```
void IloCplex::importModel(IloModel& m,
                           const char* filename,
                           IloObjective& obj,
                           IloNumVarArray vars,
                           IloRangeArray  rngs) const;
```

and

```
void IloCplex::importModel(IloModel& m,
                           const char* filename,
                           IloObjective& obj,
                           IloNumVarArray vars,
                           IloRangeArray rngs,
                           IloSOS1Array sos1,
                           IloSOS2Array sos2) const;
```

They provide additional parameters so that the newly created modeling objects will be returned to the caller. Example program `ilolpex2.cpp` gives an example of how to use method `importModel()`.

Selecting an Optimizer

`IloCplex` treats all problems it solves as Mixed Integer Programming (MIP) problems. The algorithm used by `IloCplex` for solving MIP is known as branch & cut (referred to in some contexts as branch & bound) and is described in more detail in the *ILOG CPLEX User's Manual*. For this tutorial, it is sufficient to know that this consists of solving a sequence of LPs or QPs that are generated in the course of the algorithm. The first LP or QP to be solved is known as the root, while all the others are referred to as nodes and are derived from the root or from other nodes. If the model extracted to the `cplex` object is a pure LP or QP (no integer variables), then it will be fully solved at the root.

As mentioned in *Optimizer Options* on page 13, various optimizer options are provided for solving LPs and QPs. While the default optimizer works well for a wide variety of models, `IloCplex` allows you to control which option to use for solving the root and for solving the nodes, respectively, by the following lines:

```
void IloCplex::setParam(IloCplex::RootAlg, alg)
void IloCplex::setParam(IloCplex::NodeAlg, alg)
```

where `IloCplex::Algorithm` is an enumeration type. It defines the following symbols with their meaning:

`IloCplex::AutoAlg`	allow CPLEX to choose the algorithm
`IloCplex::Dual`	use the dual simplex algorithm
`IloCplex::Primal`	use the primal simplex algorithm
`IloCplex::Barrier`	use the barrier algorithm
`IloCplex::Network`	use the network simplex algorithm for the embedded network
`IloCplex::Sifting`	use the sifting algorithm
`IloCplex::Concurrent`	allow CPLEX to use multiple algorithms on multiple computer processors

For QP models, only the `AutoAlg`, `Dual`, `Primal`, `Barrier`, and `Network` algorithms are applicable.

The optimizer option used for solving pure LPs and QPs is controlled by setting the root algorithm parameter. This is demonstrated next, in example `ilolpex2.cpp`.

Reading a Problem from a File: Example ilolpex2.cpp

This example shows how to read an optimization problem from a file, and solve it with a specified optimizer option. It prints solution information, including a Simplex basis, if available. Finally it prints the maximum infeasibility of any variable of the solution.

The file to read and the optimizer choice are passed to the program via command line parameters. For example, this command:

```
ilolpex2 example.mps d
```

reads the file `example.mps` and solves the problem with the dual simplex optimizer.

Example `ilolpex2` demonstrates:

- Reading the Model from a File
- Selecting the Optimizer
- Accessing Basis Information
- Querying Quality Measures

The general structure of this example is the same as for example `ilolpex1.cpp`. It starts by creating the environment and terminates with destroying it by calling the `end()` method. The code in between is enclosed in `try/catch` statements for error handling.

Reading the Model from a File

The model is created by reading it from the file specified as the first command line argument `argv[1]`. This is done using the method `importModel()` of an `IloCplex` object. Here the `IloCplex` object is used as a model reader rather than an optimizer. Calling `importModel()` does not extract the model to the invoking `cplex` object. This must be done later by calling `cplex.extract(model)`. We pass objects `obj`, `var`, and `rng` to `importModel()` to be able to access the variables later on when querying results.

Selecting the Optimizer

The selection of the optimizer option is done in the switch statement controlled by the second command line parameter. A call to `setParam(IloCplex::RootAlg, alg)` selects the desired `IloCplex::Algorithm` option.

Accessing Basis Information

After solving the model by calling method `solve()`, the results are accessed in the same way as in `ilolpex1.cpp`, with the exception of basis information for the variables. It is important to understand that not all optimizer options compute basis information, and thus it cannot be queried in all cases. In particular, basis information is not available when the model is solved using the barrier optimizer (`IloCplex::Barrier`) without crossover (parameter `IloCplex::BarCrossAlg` set to `IloCplex::NoAlg`).

Querying Quality Measures

Finally, the program prints the maximum primal infeasibility or bound violation of the solution. To cope with the finite precision of the numerical computations done on the computer, `IloCplex` allows some tolerances by which (for instance) optimality conditions may be violated. A long list of other quality measures is available.

Complete Program

The complete program follows. You can also view it online in the file `ilolpex2.cpp`.

```
// ------------------------------------------------------------- -*- C++ -*-
// File: examples/src/ilolpex2.cpp
// Version 8.1
// --------------------------------------------------------------------------
//  Copyright (C) 1999-2002 by ILOG.
//  All Rights Reserved.
//  Permission is expressly granted to use this example in the
//  course of developing applications that use ILOG products.
// --------------------------------------------------------------------------
//
// ilolpex2.cpp - Reading in and optimizing a problem
//
// To run this example, command line arguments are required.
// i.e.,   ilolpex2   filename   method
// where
//     filename is the name of the file, with .mps, .lp, or .sav extension
//     method   is the optimization method
//                 o          default
//                 p          primal simplex
//                 d          dual   simplex
//                 h          barrier with crossover
//                 b          barrier without crossover
//                 n          network with dual simplex cleanup
//                 s          sifting
//                 c          concurrent
// Example:
//     ilolpex2  example.mps  o
//

#include <ilcplex/ilocplex.h>
ILOSTLBEGIN

static void usage (const char *progname);

int
main (int argc, char **argv)
{
   IloEnv env;
   try {
      IloModel model(env);
```

```
IloCplex cplex(env);

if (( argc != 3 )                                ||
    ( strchr ("podhbnsc", argv[2][0]) == NULL )  ) {
   usage (argv[0]);
   throw(-1);
}

switch (argv[2][0]) {
   case 'o':
      cplex.setParam(IloCplex::RootAlg, IloCplex::AutoAlg);
      break;
   case 'p':
      cplex.setParam(IloCplex::RootAlg, IloCplex::Primal);
      break;
   case 'd':
      cplex.setParam(IloCplex::RootAlg, IloCplex::Dual);
      break;
   case 'b':
      cplex.setParam(IloCplex::RootAlg, IloCplex::Barrier);
      cplex.setParam(IloCplex::BarCrossAlg, IloCplex::NoAlg);
      break;
   case 'h':
      cplex.setParam(IloCplex::RootAlg, IloCplex::Barrier);
      break;
   case 'n':
      cplex.setParam(IloCplex::RootAlg, IloCplex::Network);
      break;
   case 's':
      cplex.setParam(IloCplex::RootAlg, IloCplex::Sifting);
      break;
   case 'c':
      cplex.setParam(IloCplex::RootAlg, IloCplex::Concurrent);
      break;
   default:
      break;
}

IloObjective   obj;
IloNumVarArray var(env);
IloRangeArray  rng(env);
cplex.importModel(model, argv[1], obj, var, rng);

cplex.extract(model);
if ( !cplex.solve() ) {
   env.error() << "Failed to optimize LP" << endl;
   throw(-1);
}

IloNumArray vals(env);
cplex.getValues(vals, var);
env.out() << "Solution status = " << cplex.getStatus() << endl;
```

```
      env.out() << "Solution value  = " << cplex.getObjValue() << endl;
      env.out() << "Solution vector = " << vals << endl;

      try {     // basis may not exist
         IloCplex::BasisStatusArray cstat(env);
         cplex.getBasisStatuses(cstat, var);
         env.out() << "Basis statuses  = " << cstat << endl;
      } catch (...) {
      }

      env.out() << "Maximum bound violation = "
                << cplex.getQuality(IloCplex::MaxPrimalInfeas) << endl;
   }
   catch (IloException& e) {
      cerr << "Concert exception caught: " << e << endl;
   }
   catch (...) {
      cerr << "Unknown exception caught" << endl;
   }

   env.end();
   return 0;
}  // END main


static void usage (const char *progname)
{
   cerr << "Usage: " << progname << " filename algorithm" << endl;
   cerr << "   where filename is a file with extension " << endl;
   cerr << "      MPS, SAV, or LP (lower case is allowed)" << endl;
   cerr << "   and algorithm is one of the letters" << endl;
   cerr << "      o          default" << endl;
   cerr << "      p          primal simplex" << endl;
   cerr << "      d          dual simplex  " << endl;
   cerr << "      b          barrier       " << endl;
   cerr << "      h          barrier with crossover" << endl;
   cerr << "      n          network simplex" << endl;
   cerr << "      s          sifting" << endl;
   cerr << "      c          concurrent" << endl;
   cerr << " Exiting..." << endl;
} // END usage
```

Modifying and Reoptimizing

In many situations, the solution to a model is only the first step. One of the important features of Concert Technology is the ability to modify and then re-solve the model even after it has been extracted and solved one or more times.

A look back to examples `ilolpex1.cpp` and `ilolpex2.cpp` reveals that we have been performing modification operations on models all along. Each time we add an extractable to a model we are changing the model. However, those examples made all such changes before the model was extracted to `cplex`.

Concert Technology maintains a link between the model and all `IloCplex` objects that may have extracted it. This link is known as *notification*. Each time a modification of the model or one of its extractables occurs, the change is notified to the `IloCplex` objects that extracted the model. They then track the modification in their internal representations.

Moreover, `IloCplex` tries to maintain as much information from a previous solution as is possible and reasonable, when the model is modified, in order to have a better start when solving the modified model. In particular, when solving LPs or QPs with a simplex method, `IloCplex` attempts to maintain a basis which will be used the next time method `solve()` is invoked, with the aim of making subsequent solves go faster.

Modifying an Optimization Problem: Example ilolpex3.cpp

This example demonstrates:

- Setting CPLEX Parameters
- Modifying an Optimization Problem
- Starting from a Previous Basis

Here is the problem example `ilolpex3` solves:

Minimize c^*x

subject to $Hx = d$

$Ax = b$

$1 \leq x \leq u$

where

$$H = \begin{pmatrix} -1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & -1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & -1 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & 0 & -1 & 0 & 1 & -1 \end{pmatrix} \quad d = \begin{pmatrix} -3 \\ 1 \\ 4 \\ 3 \\ -5 \end{pmatrix}$$

$$A = \begin{pmatrix} 2 & 1 & -2 & -1 & 2 & -1 & -2 & -3 \\ 1 & -3 & 2 & 3 & -1 & 2 & 1 & 1 \end{pmatrix} \quad b = \begin{pmatrix} 4 \\ -2 \end{pmatrix}$$

```
c =      (-9  1  4  2 -8  2  8 12 )
l =      ( 0  0  0  0  0  0  0  0 )
u =      (50 50 50 50 50 50 50 50 )
```

The constraints $Hx=d$ represent a pure network flow. The example solves this problem in two steps:

1. The CPLEX Network Optimizer is used to solve

Minimize	$c*x$
subject to	$Hx = d$
	$1 \leq x \leq u$

2. The constraints $Ax=b$ are added to the problem, and the dual simplex optimizer is used to solve the full problem, starting from the optimal basis of the network problem. The dual simplex method is highly effective in such a case because this basis remains dual feasible after the slacks (artificial variables) of the added constraints are initialized as basic.

Notice that the `0` values in the data are omitted in the example program. CPLEX makes extensive use of sparse matrix methods and, although CPLEX correctly handles any explicit zero coefficients given to it, most programs solving models of more than modest size benefit (in terms of both storage space and speed) if the natural sparsity of the model is exploited from the very start.

Before the model is solved, the network optimizer is selected by setting the `RootAlg` parameter to the value `IloCplex::Network`, as shown in example `ilolpex2.cpp`. The simplex display parameter `IloCplex::SimDisplay` is set so that the simplex algorithm issues logging information as it executes.

Setting CPLEX Parameters

`IloCplex` provides a variety of parameters that allow you to control the solution process. They can be categorized into boolean, integer, numerical and string parameters and are represented by the enumeration types `IloCplex::BoolParam`, `IloCplex::IntParam`, `IloCplex::NumParam`, and `IloCplex::StringParam`, respectively.

Modifying an Optimization Problem

After the simple model is solved and the resulting objective value is passed to the output channel `cplex.out()`, the remaining constraints are created and added to the model. At this time the model has already been extracted to `cplex`. As a consequence, whenever the model is modified by adding a constraint, this addition is immediately reflected in the `cplex` object via notification.

Starting from a Previous Basis

Before solving the modified problem, example `ilolpex3.cpp` sets the optimizer option to `IloCplex::Dual`, as this is the algorithm that can generally take best advantage of the optimal basis from the previous solve after the addition of constraints.

Complete Program

The complete program follows. You can also view it online in the file `ilolpex3.cpp`.

```
// ------------------------------------------------------------ -*- C++ -*-
// File: examples/src/ilolpex3.cpp
// Version 8.1
// --------------------------------------------------------------------------
//  Copyright (C) 1999-2002 by ILOG.
//  All Rights Reserved.
//  Permission is expressly granted to use this example in the
//  course of developing applications that use ILOG products.
// --------------------------------------------------------------------------
//
// ilolpex3.cpp, example of adding constraints to solve a problem
//
// Modified example from Chvatal, "Linear Programming", Chapter 26.
//   minimize  c*x
//   subject to  Hx = d
//               Ax = b
//               l <= x <= u
//   where
//
//   H = ( -1  0  1  0  1  0  0  0 )  d = ( -3 )
//       (  1 -1  0  1  0  0  0  0 )      (  1 )
//       (  0  1 -1  0  0  1 -1  0 )      (  4 )
//       (  0  0  0 -1  0 -1  0  1 )      (  3 )
//       (  0  0  0  0 -1  0  1 -1 )      ( -5 )
//
//   A = (  2  1 -2 -1  2 -1 -2 -3 )  b = (  4 )
//       (  1 -3  2  3 -1  2  1  1 )      ( -2 )
//
//   c = ( -9  1  4  2 -8  2  8 12 )
//   l = (  0  0  0  0  0  0  0  0 )
//   u = ( 50 50 50 50 50 50 50 50 )
//
//
//
//  Treat the constraints with A as the complicating constraints, and
//  the constraints with H as the "simple" problem.
//
//  The idea is to solve the simple problem first, and then add the
//  constraints for the complicating constraints, and solve with dual.
```

```
#include <ilcplex/ilocplex.h>
ILOSTLBEGIN

int main()
{
   IloEnv   env;
   try {
      IloModel model(env, "chvatal");

      IloNumVarArray x(env, 8, 0, 50);
      model.add(IloMinimize(env,-9*x[0] + x[1] + 4*x[2] + 2*x[3]
                               -8*x[4] + 2*x[5] + 8*x[6] + 12*x[7]));
      model.add(-x[0]        + x[2]        + x[4]                       == -3);
      model.add( x[0] - x[1]        + x[3]                             ==  1);
      model.add(        x[1] - x[2]               + x[5] - x[6]        ==  4);
      model.add(                     - x[3]       - x[5]       + x[7] ==  3);
      model.add(                             - x[4]       + x[6] - x[7] == -5);

      IloCplex cplex(model);
      cplex.setParam(IloCplex::SimDisplay, 2);
      cplex.setParam(IloCplex::RootAlg, IloCplex::Network);
      cplex.solve();
      cplex.out() << "After network optimization, objective is "
                  << cplex.getObjValue() << endl;

      model.add(2*x[0] + 1*x[1] - 2*x[2] - 1*x[3] +
                2*x[4] - 1*x[5] - 2*x[6] - 3*x[7] ==  4);
      model.add(1*x[0] - 3*x[1] + 2*x[2] + 3*x[3] -
                1*x[4] + 2*x[5] + 1*x[6] + 1*x[7] == -2);

      cplex.setParam(IloCplex::RootAlg, IloCplex::Dual);
      cplex.solve();

      IloNumArray vals(env);
      cplex.getValues(vals, x);
      cplex.out() << "Solution status " << cplex.getStatus() << endl;
      cplex.out() << "Objective value " << cplex.getObjValue() << endl;
      cplex.out() << "Solution is: " << vals << endl;

      cplex.exportModel("lpex3.sav");
   }
   catch (IloException& e) {
      cerr << "Concert exception caught: " << e << endl;
   }
   catch (...) {
      cerr << "Unknown exception caught" << endl;
   }

   env.end();
   return 0;
} // END main
```

C H A P T E R

4

Concert Technology for Java Users

This chapter is an introduction to using ILOG CPLEX through Concert Technology in the Java programming language. It gives you an overview of a typical application program, and describes procedures for:

- Creating a model
- Solving that model
- Querying results after solving
- Handling error conditions

ILOG Concert Technology allows your application to call CPLEX directly, through the JNI (Java Native Interface). This Java interface supplies a rich functionality allowing you to use Java objects to build your optimization model.

The `IloCplex` class implements the Concert Technology interface for creating variables and constraints. It also provides functionality for solving Mathematical Programing (MP) problems and accessing solution information.

Compiling CPLEX Applications in Concert Technology

When compiling a Java program that uses ILOG CPLEX Concert Technology, you need to inform the Java compiler where to find the file `cplex.jar` containing the ILOG CPLEX

Concert Technology class library. To do this, you add the `cplex.jar` file to your classpath. This is most easily done by passing the command-line option

```
-classpath <path_to_cplex.jar>
```

to the Java compiler `javac`. If you need to include other Java class libraries, you should add the corresponding `jar` files to the classpath as well. Ordinarily, you should also include the current directory '`.`' to be part of the Java classpath.

At execution time, the same classpath setting is needed. Additionally, since CPLEX is implemented via JNI, you need to instruct the Java Virtual Machine (JVM) where to find the shared library (or dynamic link library) containing the native code to be called from Java. This may be done with the command line option

```
-Djava.library.path=<path_to_shared_library>
```

to the `java` command. Note that, unlike the `cplex.jar` file, the shared library is system dependent; thus the exact pathname, of the location for the library to be used, differs depending on the platform you are using.

Pre-configured compilation and runtime commands are provided in the standard distribution, through the Unix makefiles and Windows "`javamake`" file for `Nmake`. However, these scripts presume a certain relative location for the files mentioned above, and for application development most users will have their source files in some other location.

Below are suggestions for establishing build procedures for your application.

1. First check the `readme.html` file in the standard distribution, under the *Supported Platforms* heading to locate the `<machine>` and `<libformat>` entry for your Unix platform, or the compiler and library format combination for Windows.
2. Go to the subdirectory under the `examples` directory where CPLEX is installed on your machine. On Unix this will be `<machine>/<libformat>`, and on Windows it will be `<compiler>\<libformat>`. This subdirectory will contain a `makefile` or `javamake` appropriate for your platform.
3. Then use these files to compile the examples that came in the standard distribution by calling `make execute_java` (Unix) or `nmake -f javamake execute` (Windows).
4. Carefully note the locations of the needed files, both during compilation and at run time, and convert the relative path names to absolute path names for use in your own working environment.

In Case Problems Arise

If a problem occurs in the compilation phase, make sure your java compiler is correctly set up and that your classpath includes the `cplex.jar` file.

If compilation is successful and the problem occurs when executing your application, there are three likely causes:

1. If you get a message like `java.lang.NoClassDefFoundError` your classpath is not correctly set up. Make sure you use `-classpath <path_to_cplex.jar>` in your `java` command.
2. If you get a message like `java.lang.UnsatisfiedLinkError` you need to set up the path correctly so that the JVM can locate the CPLEX shared library. Make sure you use `-Djava.library.path=<path_to_shared_library>` in your `java` command.
3. If you get a message like ilm: `CPLEX: no license found for this product` or `ilm: CPLEX: invalid encrypted key "MNJVUXTDJV82" in "/usr/ilog/ilm/ access.ilm" run ilmcheck` then there is a problem with your license to use ILOG CPLEX. Review the *ILOG License Manager User's Guide and Reference* to see whether you can correct the problem. If you have verified your system and license setup but continue to experience problems, contact ILOG Technical Support and report the error messages.

The Design of CPLEX in Concert Technology

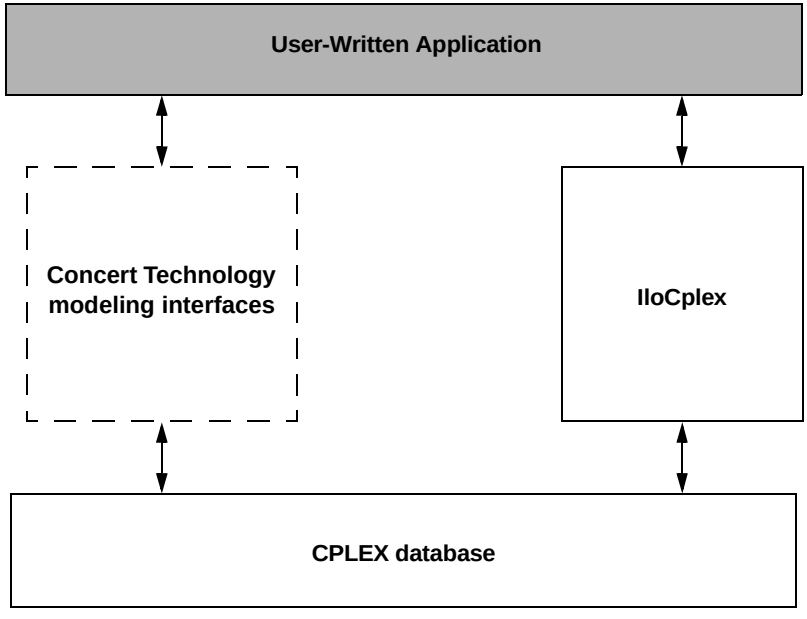

Figure 4.1 *A View of CPLEX in Concert Technology*

Figure 4.1 illustrates the design of Concert Technology and how a user program uses it. Concert Technology defines a set of interfaces for modelling objects. Such interfaces do not

actually consume memory (this is the reason the box in the figure has a dotted outline). When creating a Concert Technology modelling object using CPLEX, an object is created in the CPLEX database that implements the interface defined by Concert Technology. However, a user application never accesses such objects directly but only communicates with them through the interfaces defined by Concert Technology.

The only Concert Technology objects directly created and accessed by a user are objects from class `IloCplex`. This class implements two interfaces, `IloModeler` and `IloMPModeler`, that allow you to create modelling objects. The class `IloCplex` also provides methods to solve models and query solutions.

The Anatomy of a Concert Technology Application

To use the CPLEX Java interfaces, you need to import the appropriate packages into your application. This is done with the lines:

```
import ilog.concert.*;
import ilog.cplex.*;
```

As for every Java application, a CPLEX application is implemented as a method of a class. In this discussion, we will assume the method to be the static `main` method. The first task is to create an `IloCplex` object. It is used to create all the modeling objects needed to represent the model. For example, an integer variable with bounds 0 and 10 is created by calling `cplex.intVar(0, 10)`, where "`cplex`" is the `IloCplex` object.

Since Java error handling in CPLEX is done using exceptions, you should include the Concert Technology part of an application in a `try`/`catch` statement. All the exceptions thrown by any Concert Technology method are derived from `IloException`. Thus `IloException` should be caught in the `catch` statement.

In summary, here is the structure of a Java application that calls CPLEX:

```
import ilog.concert.*;
import ilog.cplex.*;
static public class Application {
  static public main(String[] args) {
     try {
       IloCplex cplex = new IloCplex();
       // create model and solve it
     } catch (IloException e) {
        System.err.println("Concert exception caught: " + e);
     }
   }
 }
```

Create the Model

The `IloCplex` object provides the functionality to create an optimization model that can be solved with `IloCplex`. The interface functions for doing so are defined by the Concert Technology interface `IloModeler` and its extension `IloMPModeler`. These interfaces define the constructor functions for modeling objects of the following types, which can be used with `IloCplex`:

`IloNumVar`	modeling variables
`IloRange`	ranged constraints of the type lb <= expr <= ub
`IloObjective`	optimization objective
`IloNumExpr`	expression using variables

Modeling variables are represented by objects implementing the `IloNumVar` interface defined by Concert Technology. Here is how to create three continuous variables, all with bounds `0` and `100`:

```
IloNumVar[] x = cplex.numVarArray(3, 0.0, 100.0);
```

There is a wealth of other functions for creating arrays or individual modeling variables. The documentation for `IloModeler` and `IloMPModeler` will give you the complete list.

Modeling variables are typically used to build expressions, of type `IloNumExpr`, for use in constraints or the objective function of an optimization model. For example the expression:

```
x[0] + 2*x[1] + 3*x[2]
```

can be created like this:

```
IloNumExpr expr = cplex.sum(x[0], cplex.prod(2.0, x[1]),
                            cplex.prod(3.0, x[2]));
```

Another way of creating an object representing the same expression is to use an `IloLinearNumExpr` expression. Here is how:

```
IloLinearNumExpr expr = cplex.linearNumExpr();
expr.addTerm(1.0, x[0]);
expr.addTerm(2.0, x[1]);
expr.addTerm(3.0, x[2]);
```

The advantage of using `IloLinearNumExpr` over the first way is that you can more easily build up your linear expression in a loop, which is what is typically needed in more complex applications. Interface `IloLinearNumExpr` is an extension of `IloNumExpr`, and thus can be used anywhere an expression can be used.

As mentioned before, expressions can be used to create constraints or an objective function for a model. Here is how to create a minimization objective for the above expression:

```
IloObjective obj = cplex.minimize(expr);
```

In addition to creating an objective, `IloCplex` must be instructed to use it in the model it solves. This is done by adding the objective to `IloCplex` via:

```
cplex.add(obj);
```

Every modeling object that is to be used in a model must be added to the `IloCplex` object. The variables need not be explicitly added as they are treated implicitly when used in the expression of the objective. More generally, every modeling object that is referenced by another modeling object which itself has been added to `IloCplex`, is implicitly added to `IloCplex` as well.

There is a shortcut notation for creating and adding the objective to `IloCplex`:

```
cplex.addMinimize(expr);
```

Since we don't need to access the objective otherwise, we can avoid storing it in variable `obj`.

Adding constraints to the model is just as easy. For example, the constraint

```
-x[0] + x[1] + x[2] <= 20.0
```

can be added by calling:

```
cplex.addLe(cplex.sum(cplex.negative(x[0]), x[1], x[2]), 20);
```

Again, many methods are provided for adding other constraint types, including equality constraints, greater than or equal to constraints, and ranged constraints. Internally, they are all represented as `IloRange` objects with appropriate choices of bounds, which is why all these methods return `IloRange` objects. Also, note that the expressions above could have been created in many different ways, including the use of `IloLinearNumExpr`.

Solve the Model

So far we have discussed some methods of `IloCplex` for creating models. All such methods are defined in the interfaces `IloModeler` and its extension `IloMPModeler`. However, `IloCplex` not only implements these interfaces but also provides additional methods for solving a model and querying its results.

After a model has been created as described in the previous section, the `IloCplex` object `cplex` is ready to solve the model, which consists of all the modeling objects that have been added to it. Invoking the optimizer is as simple as calling method `solve`.

Method `solve` returns a Boolean indicating whether the optimization succeeded in finding a solution. If no solution was found, `false` is returned. If `true` is returned, then CPLEX found a feasible solution, though it is not necessarily an optimal solution. More precise information about the outcome of the last call to method `solve` can be obtained by calling:

```
IloCplex.getStatus();
```

The returned value tells you what CPLEX found out about the model: whether it found the optimal solution or only a feasible solution, whether it proved the model to be unbounded or

infeasible, or whether nothing at all has been determined at this point. Even more detailed information about the termination of the solver call is available through the method:

```
IloCplex.getCplexStatus();
```

Query the Results

If the `solve` method succeeded in finding a solution, you will then want to access that solution. The objective value of that solution can be queried using method:

```
double objval = cplex.getObjValue();
```

Similarly, solution values for all the variables in the array `x` can be queried by calling:

```
double[] xval = cplex.getValues(x);
```

More solution information can be queried from `IloCplex`, including slacks and, depending on the algorithm that was applied for solving the model, duals, reduced cost information, and basis information.

Building and Solving a Small LP Model in Java

The example `LPex1.java`, which is distributed with the installation files for CPLEX, is a program that builds a specific small LP model and then solves it. This example follows the general structure found in many CPLEX Concert Technology applications, and demonstrates three main ways to construct a model:

- modeling by rows,
- modeling by columns, and
- modeling by nonzero elements.

Example `LPex1.java` is an extension of the example presented on page 34:

Maximize	$x_1 + 2x_2 + 3x_3$
subject to	$-x_1 + x_2 + x_3 \leq 20$ $x_1 - 3x_2 + x_3 \leq 30$
with these bounds	$0 \leq x_1 \leq 40$ $0 \leq x_2 \leq +\infty$ $0 \leq x_3 \leq +\infty$

After an initial check that a valid option string was provided as a calling argument, the program begins by enclosing all executable statements that follow in a `try/catch` pair of statements. In case of an error CPLEX Concert Technology will throw an exception of type `IloException`, which the catch statement then processes. In this simple example, an

exception triggers the printing of a line stating “Concert exception ‘e’ caught”, where ‘e’ is the specific exception.

We first create the model object `cplex` by executing the statement

```
IloCplex cplex = new IloCplex();
```

At this point, the cplex object represents an empty model, that is a model with no variables, constraints or other content. The model is then populated in one of several ways depending on the command line argument. The possible choices are implemented in the methods

- `populateByRow`
- `populateByColumn`
- `populateByNonzero`

All these methods pass the same three arguments. The first argument is the `cplex` object to be populated. The second and third arguments correspond to the variables (`var`) and range constraints (`rng`) respectively; the methods will write to `var[0]` and `rng[0]` an array of all the variables and constraints in the model, for later access.

After the model has been created in the `cplex` object, it is ready to be solved by calling `cplex.solve()`. The solution log will be output to the screen; this is because `IloCplex` prints all logging information to the `OutputStream cplex.out()`, which by default is initialized to `System.out`. You can change this by calling the method `cplex.setOut()`. In particular, you can turn off logging by setting the output stream to `null`, i.e. by calling `cplex.setOut(null)`. Similarly, `IloCplex` issues warning messages to `cplex.warning()`, and `cplex.setWarning()` can be used to change (or turn off) the `OutputStream` that will be used.

If the `solve()` method finds a solution for the active model, it returns `true` and we enter the section of code that accesses the solution. The method `cplex.getValues(var[0])` returns an array of primal solution values for all the variables. This array is stored as `double[] x`. The values in `x` are ordered such that `x[j]` is the primal solution value for variable `var[0][j]`. Similarly, the reduced costs, duals, and slack values are queried and stored in arrays `dj`, `pi`, and `slack`, respectively. Finally, the solution status of the active model and the objective value of the solution are queried with the methods `IloCplex.getStatus()` and `IloCplex.getObjValue()`, respectively. The program then concludes by printing the values that have been obtained in the previous steps, and terminates after calling `cplex.end()` to free the memory used by the model object; the `catch` method of `IloException` provides screen output in case of any error conditions along the way.

The remainder of the example source code is devoted to the details of populating the model object, mentioned above, and the following three sections provide details on how the methods work.

Modeling by Rows

The method `populateByRow` creates the model by adding the finished constraints and objective function to the active model, one by one. It does so by first creating the variables with the method `cplex.numVarArray()`. Then the minimization objective function is created, and added to the active model, with the method `IloCplex.addMinimize()`. The expression that defines the objective function is created by a method, `IloCplex.scalProd()`, that forms a scalar product using an array of objective coefficients times the array of variables. Finally, each of the two constraints of the model are created and added to the active model with the method `IloCplex.addLe`. For building the constraint expression, the methods `IloCplex.sum()` and `IloCplex.prod()` are used, as a contrast to the approach used in constructing the objective function.

Modeling by Columns

While for many examples population by rows may seem most straightforward and natural, there are some models where population by columns is a more natural or more efficient approach to implement. For example problems with network structure typically lend themselves well to modeling by column. Readers familiar with matrix algebra may view the method `populateByColumn()` as the transpose of `populateByRow()`.

Range objects are created for modeling by column with only their lower and upper bound. No expressions are given — building them at this point would be impossible since the variables have not been created yet. Similarly, the objective function is created only with its intended optimization sense, and without any expression.

Next the variables are created and installed in the existing ranges and objective. These newly created variables are introduced into the ranges and the objective by means of column objects, which are implemented in the class `IloColumn`. Objects of this class are created with the methods `IloCplex.column()`, and can be linked together with the method `IloColumn.and()` to form aggregate `IloColumn` objects.

An `IloColumn` object created with the method `IloCplex.column()` contains information about how to use this column to introduce a new variable into an existing modeling object. For example if `obj` is an `IloObjective` object, `cplex.column(obj, 2.0)` creates an `IloColumn` object containing the information to install a new variable in the expression of the `IloObjective` object `obj` with a linear coefficient of `2.0`. Similarly, for an `IloRange` constraint `rng`, the method call `cplex.column(rng, -1.0)` creates an `IloColumn` object containing the information to install a new variable into the expression of `rng`, as a linear term with coefficient `-1.0`.

When using a modeling by column approach, new columns are created and installed as variables in all existing modeling objects where they are needed. To do this with Concert Technology you create an `IloColumn` object for every modeling object in which you want

to install a new variable, and link them together with the method `IloColumn.and()`. For example the first variable in `populateByColumn` is created like this:

```
var[0][0] = model.numVar(model.column(obj,  1.0).and(
                         model.column(r0,  -1.0).and(
                         model.column(r1,   1.0))),
                         0.0, 40.0);
```

The three methods `model.column()` create `IloColumn` objects for installing a new variable in the objective `obj` and in the constraints `r0` and `r1`, with linear coefficients `1.0`, `-1.0`, and `1.0`, respectively. They are all linked to an aggregate column object using the method `and()`. This aggregate column object is passed as the first argument to the method `numVar()`, along with the bounds `0.0` and `40.0` as the other two arguments. The method `numVar` now creates a new variable and immediately installs it in the modeling objects `obj`, `r0`, and `r1` as described by the aggregate column object. Once installed, the new variable is returned and stored in `var[0][0]`.

Modeling by Nonzeros

The last of the three functions for building the model is `populateByNonzero()`. In this function we create the variables with only their bounds, and the empty constraints, that is, ranged constraints only with lower and upper bound but with no expression. Only after that do we construct the expressions, in a manner similar to the ones already described, using these existing variables and install them in the existing constraints with the method `IloRange.setExpr()`.

Complete Code of LPex1.java

```
// --------------------------------------------------------------------------
// File: examples/src/LPex1.java
// Version 8.1
// --------------------------------------------------------------------------
//  Copyright (C) 2001-2002 by ILOG.
//  All Rights Reserved.
//  Permission is expressly granted to use this example in the
//  course of developing applications that use ILOG products.
// --------------------------------------------------------------------------
//
// LPex1.java - Entering and optimizing an LP problem
//
// Demonstrates different methods for creating a problem.  The user has to
// choose the method on the command line:
//
//    java LPex1  -r     generates the problem by adding constraints
//    java LPex1  -c     generates the problem by adding variables
//    java LPex1  -n     generates the problem by adding expressions
//

import ilog.concert.*;
```

```
import ilog.cplex.*;


public class LPex1 {
  static void usage() {
    System.out.println("usage:   LPex1 <option>");
    System.out.println("options:       -r   build model row by row");
    System.out.println("options:       -c   build model column by column");
    System.out.println("options:       -n   build model nonzero by nonzero");
  }

  public static void main(String[] args) {
    if ( args.length != 1 || args[0].charAt(0) != '-' ) {
      usage();
      return;
    }

    try {
      // Create the modeler/solver object
      IloCplex cplex = new IloCplex();

      IloNumVar[][] var = new IloNumVar[1][];
      IloRange[][]  rng = new IloRange[1][];

      // Evaluate command line option and call appropriate populate method.
      // The created ranges and variables are returned as element 0 of arrays
      // var and rng.
      switch ( args[0].charAt(1) ) {
      case 'r': populateByRow(cplex, var, rng);
                break;
      case 'c': populateByColumn(cplex, var, rng);
                break;
      case 'n': populateByNonzero(cplex, var, rng);
                break;
      default:  usage();
                return;
      }

      // write model to file
      cplex.exportModel("lpex1.lp");

      // solve the model and display the solution if one was found
      if ( cplex.solve() ) {
        double[] x     = cplex.getValues(var[0]);
        double[] dj    = cplex.getReducedCosts(var[0]);
        double[] pi    = cplex.getDuals(rng[0]);
        double[] slack = cplex.getSlacks(rng[0]);

        cplex.out().println("Solution status = " + cplex.getStatus());
        cplex.out().println("Solution value  = " + cplex.getObjValue());

        int ncols = cplex.getNcols();
        for (int j = 0; j < ncols; ++j) {
           cplex.out().println("Column: " + j +
                               " Value = " + x[j] +
```

```
                              " Reduced cost = " + dj[j]);
      }

      int nrows = cplex.getNrows();
      for (int i = 0; i < nrows; ++i) {
         cplex.out().println("Row   : " + i +
                              " Slack = " + slack[i] +
                              " Pi = " + pi[i]);
      }
    }
    cplex.end();
  }
  catch (IloException e) {
    System.err.println("Concert exception '" + e + "' caught");
  }
}


// The following methods all populate the problem with data for the following
// linear program:
//
//    Maximize
//     x1 + 2 x2 + 3 x3
//    Subject To
//     - x1 + x2 + x3 <= 20
//     x1 - 3 x2 + x3 <= 30
//    Bounds
//     0 <= x1 <= 40
//    End
//
// using the IloMPModeler API

static void populateByRow(IloMPModeler model,
                          IloNumVar[][] var,
                          IloRange[][] rng) throws IloException {
  double[]    lb = {0.0, 0.0, 0.0};
  double[]    ub = {40.0, Double.MAX_VALUE, Double.MAX_VALUE};
  IloNumVar[] x  = model.numVarArray(3, lb, ub);
  var[0] = x;

  double[] objvals = {1.0, 2.0, 3.0};
  model.addMaximize(model.scalProd(x, objvals));

  rng[0] = new IloRange[2];
  rng[0][0] = model.addLe(model.sum(model.prod(-1.0, x[0]),
                                    model.prod( 1.0, x[1]),
                                    model.prod( 1.0, x[2])), 20.0);
  rng[0][1] = model.addLe(model.sum(model.prod( 1.0, x[0]),
                                    model.prod(-3.0, x[1]),
                                    model.prod( 1.0, x[2])), 30.0);
}

static void populateByColumn(IloMPModeler model,
                             IloNumVar[][] var,
                             IloRange[][] rng) throws IloException {
```

```
    IloObjective obj = model.addMaximize();

    rng[0] = new IloRange[2];
    rng[0][0] = model.addRange(-Double.MAX_VALUE, 20.0);
    rng[0][1] = model.addRange(-Double.MAX_VALUE, 30.0);

    IloRange r0 = rng[0][0];
    IloRange r1 = rng[0][1];

    var[0] = new IloNumVar[3];
    var[0][0] = model.numVar(model.column(obj,  1.0).and(
                             model.column(r0,  -1.0).and(
                             model.column(r1,   1.0))),
                             0.0, 40.0);
    var[0][1] = model.numVar(model.column(obj,  2.0).and(
                             model.column(r0,   1.0).and(
                             model.column(r1,  -3.0))),
                             0.0, Double.MAX_VALUE);
    var[0][2] = model.numVar(model.column(obj,  3.0).and(
                             model.column(r0,   1.0).and(
                             model.column(r1,   1.0))),
                             0.0, Double.MAX_VALUE);
  }

  static void populateByNonzero(IloMPModeler model,
                                IloNumVar[][] var,
                                IloRange[][] rng) throws IloException {
    double[]    lb = {0.0, 0.0, 0.0};
    double[]    ub = {40.0, Double.MAX_VALUE, Double.MAX_VALUE};
    IloNumVar[] x  = model.numVarArray(3, lb, ub);
    var[0] = x;

    double[] objvals = {1.0, 2.0, 3.0};
    model.add(model.maximize(model.scalProd(x, objvals)));

    rng[0] = new IloRange[2];
    rng[0][0] = model.addRange(-Double.MAX_VALUE, 20.0);
    rng[0][1] = model.addRange(-Double.MAX_VALUE, 30.0);

    rng[0][0].setExpr(model.sum(model.prod(-1.0, x[0]),
                                model.prod( 1.0, x[1]),
                                model.prod( 1.0, x[2])));
    rng[0][1].setExpr(model.sum(model.prod( 1.0, x[0]),
                                model.prod(-3.0, x[1]),
                                model.prod( 1.0, x[2])));
  }
}
```

C H A P T E R

5

Callable Library Tutorial

This tutorial shows how to write programs that use the CPLEX Callable Library. In this chapter you will learn about:

- The Design of the ILOG CPLEX Callable Library
- Compiling and Linking Callable Library Applications
- How ILOG CPLEX Works
- Creating a Successful Callable Library Application
- Building and Solving a Small LP Model in C
- Reading a Problem from a File: Example lpex2.c
- Adding Rows to a Problem: Example lpex3.c
- Performing Sensitivity Analysis

The Design of the ILOG CPLEX Callable Library

Figure 5.1 shows a picture of the ILOG CPLEX world. The ILOG CPLEX Callable Library together with the ILOG CPLEX database make up the ILOG CPLEX core. The core becomes associated with your application through Callable Library routines. The ILOG

CPLEX environment and all problem-defining data are established inside the ILOG CPLEX core.

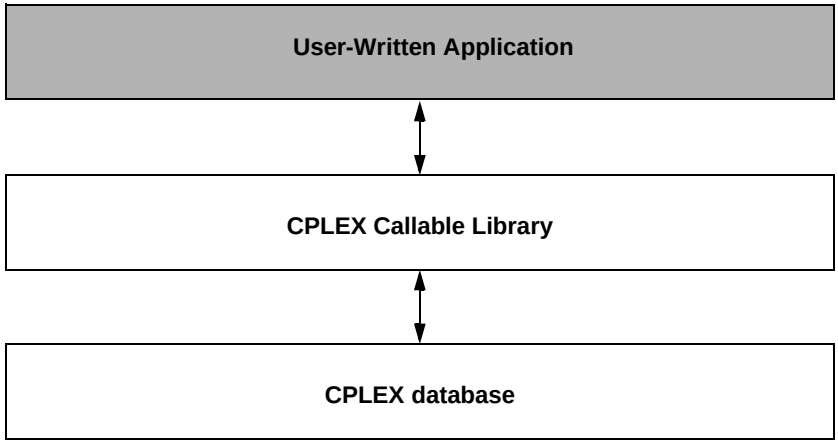

Figure 5.1 *A View of the CPLEX Callable Library*

The ILOG CPLEX Callable Library includes several categories of routines:

- optimization and result routines for defining a problem, optimizing it, and getting the results;
- utility routines for addressing application programming matters;
- problem modification routines to change a problem once it has been created within the ILOG CPLEX database;
- problem query routines to access information about a problem once it has been created;
- file reading and writing routines to move information from the file system into your application or out of your application to the file system;
- parameter setting and query routines to access and modify the values of control parameters maintained by ILOG CPLEX.

Compiling and Linking Callable Library Applications

Each Callable Library is distributed as a single library file `libcplex.a` or `cplex81.lib`. Use of the library file is similar to that with `.o` or `.obj` files. Simply substitute the library file in the link procedure. This procedure simplifies linking and ensures that the smallest possible executable is generated.

The following compilation and linking instructions assume that the example source programs and CPLEX Callable Library files are in the directories associated with a default installation of the software. If this is not true, additional compile and link flags would be required to point to the locations of the include file `cplex.h`, and Callable Library files respectively.

Note: The instructions below were current at the time of publication. As compilers, linkers and operating systems are released, different instructions may apply. Be sure to check the *Release Notes* that come with your CPLEX distribution for any changes. Also check the CPLEX web page (`http://www.ilog.com/products/cplex`).

Building CPLEX Callable Library Applications on UNIX Platforms

To compile and execute an example (`lpex1`) do the following:

```
% cd examples/<machine>/<libformat>
% make lpex1     #  to compile and execute the first CPLEX example
```

A list of all the examples that can be built this way is to be found in the makefile by looking for `C_EX` (C examples), or you can view the files listed in `examples/src`.

The makefile contains recommended compiler flags and other settings for your particular computer, which you can find by searching in it for "Compiler options" and use in your applications that call CPLEX.

Building CPLEX Callable Library Applications on Win32 Platforms

Building a CPLEX application using MS Visual C++ Integrated Development Environment, or the MS Visual C++ command line compiler are explained here.

Microsoft Visual C++ IDE

To make a CPLEX Callable Library application using Visual C++, first create or open a project in the Visual C++ Integrated Development Environment (IDE). Project files are provided for each of the examples found in the directory `examples\msvc6\<libformat>` and `examples\msvc6\<libformat>`. For details on the build process, refer to the information file `msvc.html`, which is found in the top of the installed CPLEX directory structure.

Note: *The distributed application must be able to locate* `cplex81.dll` *at run time.*

Microsoft Visual C++ Command Line Compiler

If the Visual C++ command line compiler is used outside of the IDE, the command should resemble the following example. The example command assumes that the file `cplex81.lib` is in the current directory with the source file `lpex1.c`, and that the line in

the source file "`#include <ilcplex/cplex.h>`" correctly points to the location of the include file or else has been modified to do so (or that the directories containing these files have been added to the environment variables `LIB` and `INCLUDE` respectively).

```
cl lpex1.c cplex81.lib
```

This command will create the executable file `lpex1.exe`.

Using Dynamic Loading

Some projects require more precise control over the loading and unloading of DLLs. For information on loading and unloading DLLs without using static linking, please refer to the compiler documentation or to a book such as *Advanced Windows* by Jeffrey Richter from Microsoft Press. If this is not a requirement, the static link implementations mentioned above are easier to use.

Building Applications that Use the CPLEX Parallel Optimizers

When compiling and linking programs that use the CPLEX Parallel Optimizers, it is especially important to review the relevant flags for the compiler and linker. These are found in the makefile provided with UNIX distributions or in the sample project files provided with Windows distributions. We recommend you also review the section on *Using Parallel Optimizers* in the *ILOG CPLEX User's Manual* for important details pertaining to each specific parallel optimizer.

How ILOG CPLEX Works

When your application uses routines of the ILOG CPLEX Callable Library, it must first open the ILOG CPLEX environment, then create and populate a problem object before it solves a problem. Before it exits, the application must also free the problem object and release the ILOG CPLEX environment. The following sections explain those steps.

Opening the ILOG CPLEX Environment

ILOG CPLEX requires a number of internal data structures in order to execute properly. These data structures must be initialized before any call to the ILOG CPLEX Callable Library. The first call to the ILOG CPLEX Callable Library is always to the function `CPXopenCPLEX()`. This routine checks for a valid ILOG CPLEX license and returns a pointer to the ILOG CPLEX environment. This pointer is then passed to every ILOG CPLEX Callable Library routine, except `CPXmsg()`.

The application developer must make an independent decision as to whether the variable containing the environment pointer is a global or local variable. Multiple environments are allowed, but extensive opening and closing of environments may create significant overhead on the licensor and degrade performance; typical applications make use of only one

environment for the entire execution, since a single environment may hold as many problem objects as the user wishes. After all calls to the Callable Library are complete, the environment is released by the routine `CPXcloseCPLEX()`. This routine indicates to ILOG CPLEX that all calls to the Callable Library are complete, any memory allocated by ILOG CPLEX is returned to the operating system, and the use of the ILOG CPLEX license is ended for this run.

Instantiating the Problem Object

A *problem object* is instantiated (created and initialized) by ILOG CPLEX when you call the routine `CPXcreateprob()`. It is destroyed when you call `CPXfreeprob()`. ILOG CPLEX allows you to create more than one problem object, although typical applications will use only one. Each problem object is referenced by a pointer returned by `CPXcreateprob()` and represents one specific problem instance. All Callable Library functions (except parameter setting functions and message handling functions) require a pointer to a problem object.

Populating the Problem Object

The problem object instantiated by `CPXcreateprob()` represents an empty problem that contains no data; it has zero constraints, zero variables, and an empty constraint matrix. This empty problem object must be populated with data. This step can be carried out in several ways.

- The problem object can be populated by assembling arrays of data and then calling `CPXcopylp()` to copy the data into the problem object. (For example, see *Building and Solving a Small LP Model in C* on page 111.)
- Alternatively, you can populate the problem object by sequences of calls to the routines `CPXnewcols()`, `CPXnewrows()`, `CPXaddcols()`, `CPXaddrows()`, and `CPXchgcoeflist()`; these routines may be called in any order that is convenient. (For example, see *Adding Rows to a Problem: Example lpex3.c* on page 131.)
- If the data already exist in a file using MPS format or LP format, you can use `CPXreadcopyprob()` to read the file and copy the data into the problem object. (For example, see *Reading a Problem from a File: Example lpex2.c* on page 122.)

Changing the Problem Object

A major consideration in the design of ILOG CPLEX is the need to efficiently re-optimize modified linear programs. In order to accomplish that, ILOG CPLEX must be aware of changes that have been made to a linear program since it was last optimized. Problem modification routines are available in the Callable Library.

Do not change the problem by changing the original problem data arrays and then making a call to `CPXcopylp()`. Instead, change the problem using the problem modification routines, allowing ILOG CPLEX to make use of as much solution information as possible from the solution of the problem before the modifications took place.

For example, suppose that a problem has been solved, and that the user has changed the upper bound on a variable through an appropriate call to the ILOG CPLEX Callable Library. A re-optimization would then begin from the previous optimal basis, and if that old basis were still optimal, then that information would be returned without even the need to refactor the old basis.

Creating a Successful Callable Library Application

Callable Library applications are created to solve a wide variety of problems. Each application shares certain common characteristics, regardless of its apparent uniqueness. The following steps can help you minimize development time and get maximum performance from your programs:

1. Prototype the Model
2. Identify the Routines to be Called
3. Test Procedures in the Application
4. Assemble the Data
5. Choose an Optimizer
6. Observe Good Programming Practices
7. Debug Your Program
8. Test Your Application
9. Use the Examples

Prototype the Model

Create a small version of the model to be solved. An algebraic modeling language is sometimes helpful during this step.

Identify the Routines to be Called

By separating the application into smaller parts, you can easily identify the tools needed to complete the application. Part of this process consists of identifying the Callable Library routines that will be called.

In some applications, the Callable Library is a small part of a larger program. In that case, the only ILOG CPLEX routines needed may be for:

- problem creation;
- optimizing;
- obtaining results.

In other cases the Callable Library is used extensively in the application. If so, Callable Library routines may also be needed to:

- modify the problem;
- set parameters;
- determine input and output messages and files;
- query problem data.

Test Procedures in the Application

It is often possible to test the procedures of an application in the ILOG CPLEX Interactive Optimizer with a small prototype of the model. Doing so will help identify the Callable Library routines required. The test may also uncover any flaws in procedure logic before you invest significant development effort.

Trying the ILOG CPLEX Interactive Optimizer is an easy way to determine the best optimization procedure and parameter settings.

Assemble the Data

You must decide which approach to populating the problem object is best for your application. Reading an MPS or LP file may reduce the coding effort but can increase the run-time and disk-space requirements of the program. Building the problem in memory and then calling `CPXcopylp()` avoids time consuming disk-file reading. Using the routines `CPXnewcols()`, `CPXnewrows()`, `CPXaddcols()`, `CPXaddrows()`, and `CPXchgcoeflist()` can lead to modular code that may be more easily maintained than if you assemble all model data in one step.

Another consideration is that if the Callable Library application reads an MPS or LP formatted file, usually another application is required to generate that file. Particularly in the case of MPS files, the data structures used to generate the file could almost certainly be used to build the problem-defining arrays for `CPXcopylp()` directly. The result would be less coding and a faster, more efficient application. These observations suggest that formatted files may be useful when prototyping your application, while assembling the arrays in memory may be a useful enhancement for a production version.

Choose an Optimizer

Once a problem object has been instantiated and populated, it can be solved using one of the optimizers provided by the ILOG CPLEX Callable Library. The choice of optimizer depends on the problem type.

- LP and QP problems can be solved by:
  - the primal simplex optimizer;
  - the dual simplex optimizer; and
  - the barrier optimizer;
- LP problems can also be solved by:
  - the sifting optimizer; and
  - the concurrent optimizer.

  LP problems with a substantial network, can also be solved by a special network optimizer.
- If the problem includes integer variables, branch & cut must be used.

There are also many different possible parameter settings for each optimizer. The default values will usually be the best for linear programs. Integer programming problems are more sensitive to specific settings, so additional experimentation will often be useful.

Choosing the best way to solve the problem can dramatically improve performance. For more information, refer to the sections about tuning LP performance and trouble-shooting MIP performance in the *ILOG CPLEX User's Manual*.

Observe Good Programming Practices

Using good programming practices will save development time and make the program easier to understand and modify. A list of good programming practices is provided in the *ILOG CPLEX User's Manual*.

Debug Your Program

Your program may not run properly the first time you build it. Learn to use a symbolic debugger and other widely available tools that support the creation of error-free code. Use the list of debugging tips provided in the *ILOG CPLEX User's Manual* to find and correct problems in your Callable Library application.

Test Your Application

Once an application works correctly, it still may have errors or features that inhibit execution speed. To get the most out of your application, be sure to test its performance as well as its correctness. Again, the ILOG CPLEX Interactive Optimizer can help. Since the Interactive Optimizer uses the same routines as the Callable Library, it should take the same amount of time to solve a problem as a Callable Library application.

Use the `CPXwriteprob()` routine with the SAV format to create a binary representation of the problem object, then read it in and solve it with the Interactive Optimizer. If the application sets optimization parameters, use the same settings with the Interactive Optimizer. If your application takes significantly longer than the Interactive Optimizer, performance within your application can probably be improved. In such a case, possible performance inhibitors include fragmentation of memory, unnecessary compiler and linker options, and coding approaches that slow the program without causing it to give incorrect results.

Use the Examples

The ILOG CPLEX Callable Library is distributed with a variety of examples that illustrate the flexibility of the Callable Library. The C source of all examples is provided in the standard distribution. For explanations about the examples of quadratic programming problems (QPs), mixed integer programming problems (MIPs) and network flows, see the *ILOG CPLEX User's Manual*. Explanations of the following examples of LPs appear in this manual:

- `lpex1.c` illustrates various ways of generating a problem object.
- `lpex2.c` demonstrates how to read a problem from a file, optimize it via a choice of several means, and obtain the solution.
- `lpex3.c` demonstrates how to add rows to a problem object and reoptimize.

We strongly encourage you to compile, link, and run all of the examples provided in the standard distribution.

Building and Solving a Small LP Model in C

The example `lpex1.c` shows you how to use problem modification routines from the ILOG CPLEX Callable Library in three different ways to build a model. The application in the example takes a single command line argument that indicates whether to build the constraint

matrix by rows, columns, or nonzeros. After building the problem, the application optimizes it and displays the solution. Here is the problem that the example optimizes:

Maximize	$x_1 + 2x_2 + 3x_3$
subject to	$-x_1 + x_2 + x_3 \leq 20$ $x_1 - 3x_2 + x_3 \leq 30$
with these bounds	$0 \leq x_1 \leq 40$ $0 \leq x_2 \leq +\infty$ $0 \leq x_3 \leq +\infty$

Before any ILOG CPLEX Callable Library routine can be called, your application must call the routine `CPXopenCPLEX()` to get a pointer (called `env`) to the ILOG CPLEX environment. Your application will then pass this pointer to every Callable Library routine. If this routine fails, it returns an error code. This error code can be translated to a string by the routine `CPXgeterrorstring()`.

After the ILOG CPLEX environment is initialized, the ILOG CPLEX screen indicator parameter (`CPX_PARAM_SCRIND`) is turned on by the routine `CPXsetintparam()`. This causes all default ILOG CPLEX output to appear on the screen. If this parameter is not set, then ILOG CPLEX will generate no viewable output on the screen or in a file.

At this point, the routine `setproblemdata()` is called to create an empty problem object. Based on the problem-building method selected by the command-line argument, the application then calls a routine to build the matrix by rows, by columns, or by nonzeros. The routine `populatebyrow()` first calls `CPXnewcols()` to specify the column-based problem data, such as the objective, bounds, and variables names. The routine `CPXaddrows()` is then called to supply the constraints. The routine `populatebycolumn()` first calls `CPXnewrows()` to specify the row-based problem data, such as the right-hand side values and sense of constraints. The routine `CPXaddcols()` is then called to supply the columns of the matrix and the associated column bounds, names, and objective coefficients. The routine `populatebynonzero()` calls both `CPXnewrows()` and `CPXnewcols()` to supply all the problem data except the actual constraint matrix. At this point, the rows and columns are well defined, but the constraint matrix remains empty. The routine `CPXchgcoeflist()` is then called to fill in the nonzero entries in the matrix.

Once the problem has been specified, the application optimizes it by calling the routine `CPXlpopt()`. Its default behavior is to use the ILOG CPLEX Dual Simplex Optimizer. If this routine returns a nonzero result, then an error occurred. If no error occurred, the application allocates arrays for solution values of the primal variables, dual variables, slack variables, and reduced costs; then it obtains the solution information by calling the routine `CPXsolution()`. This routine returns the status of the problem (whether optimal, infeasible, or unbounded, and whether a time limit or iteration limit was reached), the objective value and the solution vectors. The application then displays this information on the screen.

As a debugging aid, the application writes the problem to a ILOG CPLEX LP file (named `lpex1.lp`) by calling the routine `CPXwriteprob()`. This file can be examined to determine whether any errors occurred in the `setproblemdata()` or `CPXcopylp()` routines. `CPXwriteprob()` can be called at any time after `CPXcreateprob()` has created the `lp` pointer.

The label `TERMINATE:` is used as a place for the program to exit if any type of failure occurs, or if everything succeeds. In either case, the problem object represented by `lp` is released by the call to `CPXfreeprob()`, and any memory allocated for solution arrays is freed. The application then calls `CPXcloseCPLEX()`; it tells ILOG CPLEX that all calls to the Callable Library are complete. If an error occurs when this routine is called, then a call to `CPXgeterrorstring()`is needed to determine the error message, since `CPXcloseCPLEX()` causes no screen output.

Complete Program

The complete program follows. You can also view it online in the file `lpex1.c`.

```
/*--------------------------------------------------------------------------*/
/*  File: examples/src/lpex1.c                                              */
/*  Version 8.1                                                             */
/*--------------------------------------------------------------------------*/
/*  Copyright (C) 1997-2002 by ILOG.                                        */
/*  All Rights Reserved.                                                    */
/*  Permission is expressly granted to use this example in the              */
/*  course of developing applications that use ILOG products.               */
/*--------------------------------------------------------------------------*/

/* lpex1.c - Entering and optimizing a problem.  Demonstrates different
    methods for creating a problem.  The user has to choose the method
    on the command line:

      lpex1  -r     generates the problem by adding rows
      lpex1  -c     generates the problem by adding columns
      lpex1  -n     generates the problem by adding a list of coefficients
 */

/* Bring in the CPLEX function declarations and the C library
   header file stdio.h with the following single include. */

#include <ilcplex/cplex.h>
#include <stdlib.h>
#include <assert.h>
#include <math.h>
#endif

/* Bring in the declarations for the string functions */

#include <string.h>
```

```
/* Include declaration for functions at end of program */

static int
   populatebyrow     (CPXENVptr env, CPXLPptr lp),
   populatebycolumn  (CPXENVptr env, CPXLPptr lp),
   populatebynonzero (CPXENVptr env, CPXLPptr lp);

static void
   free_and_null     (char **ptr),
   usage             (char *progname);



int
main (int argc, char **argv)
{
   /* Declare and allocate space for the variables and arrays where we
      will store the optimization results including the status, objective
      value, variable values, dual values, row slacks and variable
      reduced costs. */

   int      solstat;
   double   objval;
   double   *x = NULL;
   double   *pi = NULL;
   double   *slack = NULL;
   double   *dj = NULL;


   CPXENVptr     env = NULL;
   CPXLPptr      lp = NULL;
   int           status = 0;
   int           i, j;
   int           cur_numrows, cur_numcols;

   /* Check the command line arguments */

   if (( argc != 2 )                          ||
       ( argv[1][0] != '-' )                  ||
       ( strchr ("rcn", argv[1][1]) == NULL )   ) {
      usage (argv[0]);
      goto TERMINATE;
   }

   /* Initialize the CPLEX environment */

   env = CPXopenCPLEX (&status);

   /* If an error occurs, the status value indicates the reason for
      failure.  A call to CPXgeterrorstring will produce the text of
      the error message.  Note that CPXopenCPLEX produces no output,
      so the only way to see the cause of the error is to use
      CPXgeterrorstring.  For other CPLEX routines, the errors will
```

```
   be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON.  */

if ( env == NULL ) {
   char  errmsg[1024];
   fprintf (stderr, "Could not open CPLEX environment.\n");
   CPXgeterrorstring (env, status, errmsg);
   fprintf (stderr, "%s", errmsg);
   goto TERMINATE;
}

/* Turn on output to the screen */

status = CPXsetintparam (env, CPX_PARAM_SCRIND, CPX_ON);
if ( status ) {
   fprintf (stderr,
            "Failure to turn on screen indicator, error %d.\n", status);
   goto TERMINATE;
}

/* Turn on data checking */

status = CPXsetintparam (env, CPX_PARAM_DATACHECK, CPX_ON);
if ( status ) {
   fprintf (stderr,
            "Failure to turn on data checking, error %d.\n", status);
   goto TERMINATE;
}

/* Create the problem. */

lp = CPXcreateprob (env, &status, "lpex1");

/* A returned pointer of NULL may mean that not enough memory
   was available or there was some other problem.  In the case of
   failure, an error message will have been written to the error
   channel from inside CPLEX.  In this example, the setting of
   the parameter CPX_PARAM_SCRIND causes the error message to
   appear on stdout.  */

if ( lp == NULL ) {
   fprintf (stderr, "Failed to create LP.\n");
   goto TERMINATE;
}

/* Now populate the problem with the data.  For building large
   problems, consider setting the row, column and nonzero growth
   parameters before performing this task. */

switch (argv[1][1]) {
   case 'r':
      status = populatebyrow (env, lp);
      break;
```

```
      case 'c':
         status = populatebycolumn (env, lp);
         break;
      case 'n':
         status = populatebynonzero (env, lp);
         break;
   }

   if ( status ) {
      fprintf (stderr, "Failed to populate problem.\n");
      goto TERMINATE;
   }

   /* Optimize the problem and obtain solution. */

   status = CPXlpopt (env, lp);
   if ( status ) {
      fprintf (stderr, "Failed to optimize LP.\n");
      goto TERMINATE;
   }

   /* The size of the problem should be obtained by asking CPLEX what
      the actual size is, rather than using sizes from when the problem
      was built.  cur_numrows and cur_numcols store the current number
      of rows and columns, respectively.  */

   cur_numrows = CPXgetnumrows (env, lp);
   cur_numcols = CPXgetnumcols (env, lp);

   x = (double *) malloc (cur_numcols * sizeof(double));
   slack = (double *) malloc (cur_numrows * sizeof(double));
   dj = (double *) malloc (cur_numcols * sizeof(double));
   pi = (double *) malloc (cur_numrows * sizeof(double));

   if ( x     == NULL ||
        slack == NULL ||
        dj    == NULL ||
        pi    == NULL   ) {
      status = CPXERR_NO_MEMORY;
      fprintf (stderr, "Could not allocate memory for solution.\n");
      goto TERMINATE;
   }

   status = CPXsolution (env, lp, &solstat, &objval, x, pi, slack, dj);
   if ( status ) {
      fprintf (stderr, "Failed to obtain solution.\n");
      goto TERMINATE;
   }

   /* Write the output to the screen. */

   printf ("\nSolution status = %d\n", solstat);
   printf ("Solution value  = %f\n\n", objval);
```

```
   for (i = 0; i < cur_numrows; i++) {
      printf ("Row %d:  Slack = %10f  Pi = %10f\n", i, slack[i], pi[i]);
   }

   for (j = 0; j < cur_numcols; j++) {
      printf ("Column %d:  Value = %10f  Reduced cost = %10f\n",
              j, x[j], dj[j]);
   }

   /* Finally, write a copy of the problem to a file. */

   status = CPXwriteprob (env, lp, "lpex1.lp", NULL);
   if ( status ) {
      fprintf (stderr, "Failed to write LP to disk.\n");
      goto TERMINATE;
   }


TERMINATE:

   /* Free up the solution */

   free_and_null ((char **) &x);
   free_and_null ((char **) &slack);
   free_and_null ((char **) &dj);
   free_and_null ((char **) &pi);

   /* Free up the problem as allocated by CPXcreateprob, if necessary */

   if ( lp != NULL ) {
      status = CPXfreeprob (env, &lp);
      if ( status ) {
         fprintf (stderr, "CPXfreeprob failed, error code %d.\n", status);
      }
   }

   /* Free up the CPLEX environment, if necessary */

   if ( env != NULL ) {
      status = CPXcloseCPLEX (&env);

      /* Note that CPXcloseCPLEX produces no output,
         so the only way to see the cause of the error is to use
         CPXgeterrorstring.  For other CPLEX routines, the errors will
         be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON. */

      if ( status ) {
         char  errmsg[1024];
         fprintf (stderr, "Could not close CPLEX environment.\n");
         CPXgeterrorstring (env, status, errmsg);
         fprintf (stderr, "%s", errmsg);
```

```
      }
   }

   return (status);

}  /* END main */


/* This simple routine frees up the pointer *ptr, and sets *ptr to NULL */

static void
free_and_null (char **ptr)
{
   if ( *ptr != NULL ) {
      free (*ptr);
      *ptr = NULL;
   }
} /* END free_and_null */



static void
usage (char *progname)
{
   fprintf (stderr,"Usage: %s -X\n", progname);
   fprintf (stderr,"   where X is one of the following options: \n");
   fprintf (stderr,"      r          generate problem by row\n");
   fprintf (stderr,"      c          generate problem by column\n");
   fprintf (stderr,"      n          generate problem by nonzero\n");
   fprintf (stderr," Exiting...\n");
} /* END usage */


/* These functions all populate the problem with data for the following
   linear program:

      Maximize
       obj: x1 + 2 x2 + 3 x3
      Subject To
       c1: - x1 + x2 + x3 <= 20
       c2: x1 - 3 x2 + x3 <= 30
      Bounds
       0 <= x1 <= 40
      End
 */

#define NUMROWS    2
#define NUMCOLS    3
#define NUMNZ      6


/* To populate by row, we first create the columns, and then add the
   rows.  */
```

```
static int
populatebyrow (CPXENVptr env, CPXLPptr lp)
{
   int      status    = 0;
   double   obj[NUMCOLS];
   double   lb[NUMCOLS];
   double   ub[NUMCOLS];
   char     *colname[NUMCOLS];
   int      rmatbeg[NUMROWS];
   int      rmatind[NUMNZ];
   double   rmatval[NUMNZ];
   double   rhs[NUMROWS];
   char     sense[NUMROWS];
   char     *rowname[NUMROWS];

   CPXchgobjsen (env, lp, CPX_MAX);  /* Problem is maximization */

   /* Now create the new columns.  First, populate the arrays. */

       obj[0] = 1.0;      obj[1] = 2.0;           obj[2] = 3.0;

        lb[0] = 0.0;       lb[1] = 0.0;           lb[2]  = 0.0;
        ub[0] = 40.0;      ub[1] = CPX_INFBOUND;  ub[2]  = CPX_INFBOUND;

   colname[0] = "x1"; colname[1] = "x2";      colname[2] = "x3";

   status = CPXnewcols (env, lp, NUMCOLS, obj, lb, ub, NULL, colname);
   if ( status )  goto TERMINATE;

   /* Now add the constraints.  */

   rmatbeg[0] = 0;     rowname[0] = "c1";

   rmatind[0] = 0;     rmatind[1] = 1;    rmatind[2] = 2;     sense[0] = 'L';
   rmatval[0] = -1.0;  rmatval[1] = 1.0;  rmatval[2] = 1.0;   rhs[0]   = 20.0;

   rmatbeg[1] = 3;     rowname[1] = "c2";
   rmatind[3] = 0;     rmatind[4] = 1;    rmatind[5] = 2;     sense[1] = 'L';
   rmatval[3] = 1.0;   rmatval[4] = -3.0; rmatval[5] = 1.0;   rhs[1]   = 30.0;

   status = CPXaddrows (env, lp, 0, NUMROWS, NUMNZ, rhs, sense, rmatbeg,
                        rmatind, rmatval, NULL, rowname);
   if ( status )  goto TERMINATE;

TERMINATE:

   return (status);

}  /* END populatebyrow */
```

```
/* To populate by column, we first create the rows, and then add the
   columns.  */

static int
populatebycolumn (CPXENVptr env, CPXLPptr lp)
{
   int      status    = 0;
   double   obj[NUMCOLS];
   double   lb[NUMCOLS];
   double   ub[NUMCOLS];
   char     *colname[NUMCOLS];
   int      matbeg[NUMCOLS];
   int      matind[NUMNZ];
   double   matval[NUMNZ];
   double   rhs[NUMROWS];
   char     sense[NUMROWS];
   char     *rowname[NUMROWS];

   CPXchgobjsen (env, lp, CPX_MAX);  /* Problem is maximization */

   /* Now create the new rows.  First, populate the arrays. */

   rowname[0] = "c1";
   sense[0]   = 'L';
   rhs[0]     = 20.0;

   rowname[1] = "c2";
   sense[1]   = 'L';
   rhs[1]     = 30.0;

   status = CPXnewrows (env, lp, NUMROWS, rhs, sense, NULL, rowname);
   if ( status )   goto TERMINATE;

   /* Now add the new columns.  First, populate the arrays. */

       obj[0] = 1.0;      obj[1] = 2.0;           obj[2] = 3.0;

    matbeg[0] = 0;     matbeg[1] = 2;          matbeg[2] = 4;

    matind[0] = 0;     matind[2] = 0;          matind[4] = 0;
    matval[0] = -1.0;  matval[2] = 1.0;        matval[4] = 1.0;

    matind[1] = 1;     matind[3] = 1;          matind[5] = 1;
    matval[1] = 1.0;   matval[3] = -3.0;       matval[5] = 1.0;

        lb[0] = 0.0;       lb[1] = 0.0;           lb[2]  = 0.0;
        ub[0] = 40.0;      ub[1] = CPX_INFBOUND;  ub[2]  = CPX_INFBOUND;

   colname[0] = "x1"; colname[1] = "x2";      colname[2] = "x3";

   status = CPXaddcols (env, lp, NUMCOLS, NUMNZ, obj, matbeg, matind,
                        matval, lb, ub, colname);
```

```
   if ( status )  goto TERMINATE;

TERMINATE:

   return (status);

}  /* END populatebycolumn */


/* To populate by nonzero, we first create the rows, then create the
   columns, and then change the nonzeros of the matrix 1 at a time.  */

static int
populatebynonzero (CPXENVptr env, CPXLPptr lp)
{
   int      status    = 0;
   double   obj[NUMCOLS];
   double   lb[NUMCOLS];
   double   ub[NUMCOLS];
   char     *colname[NUMCOLS];
   double   rhs[NUMROWS];
   char     sense[NUMROWS];
   char     *rowname[NUMROWS];
   int      rowlist[NUMNZ];
   int      collist[NUMNZ];
   double   vallist[NUMNZ];

   CPXchgobjsen (env, lp, CPX_MAX);  /* Problem is maximization */

   /* Now create the new rows.  First, populate the arrays. */

   rowname[0] = "c1";
   sense[0]   = 'L';
   rhs[0]     = 20.0;

   rowname[1] = "c2";
   sense[1]   = 'L';
   rhs[1]     = 30.0;

   status = CPXnewrows (env, lp, NUMROWS, rhs, sense, NULL, rowname);
   if ( status )   goto TERMINATE;

   /* Now add the new columns.  First, populate the arrays. */

       obj[0] = 1.0;      obj[1] = 2.0;           obj[2] = 3.0;

        lb[0] = 0.0;       lb[1] = 0.0;           lb[2]  = 0.0;
        ub[0] = 40.0;      ub[1] = CPX_INFBOUND;  ub[2]  = CPX_INFBOUND;

   colname[0] = "x1"; colname[1] = "x2";      colname[2] = "x3";

   status = CPXnewcols (env, lp, NUMCOLS, obj, lb, ub, NULL, colname);
```

```
   if ( status )  goto TERMINATE;

   /* Now create the list of coefficients */

   rowlist[0] = 0;   collist[0] = 0;   vallist[0] = -1.0;
   rowlist[1] = 0;   collist[1] = 1;   vallist[1] = 1.0;
   rowlist[2] = 0;   collist[2] = 2;   vallist[2] = 1.0;
   rowlist[3] = 1;   collist[3] = 0;   vallist[3] = 1.0;
   rowlist[4] = 1;   collist[4] = 1;   vallist[4] = -3.0;
   rowlist[5] = 1;   collist[5] = 2;   vallist[5] = 1.0;

   status = CPXchgcoeflist (env, lp, 6, rowlist, collist, vallist);

   if ( status )  goto TERMINATE;

TERMINATE:

   return (status);

}  /* END populatebynonzero */
```

Reading a Problem from a File: Example `lpex2.c`

The previous example, `lpex1.c` shows a way to copy problem data into a ILOG CPLEX problem object as part of an application that calls routines from the ILOG CPLEX Callable Library. Frequently, however, a file already exists containing a linear programming problem in the industry standard MPS format, the ILOG CPLEX LP format, or the ILOG CPLEX binary SAV format. In example `lpex2.c`, ILOG CPLEX file-reading and optimization routines read such a file to solve the problem.

Example `lpex2.c` uses command line arguments to determine the name of the input file and the optimizer to call.

Usage: `lpex2 filename optimizer`

Where: `filename` is a file with extension MPS, SAV, or LP (lower case is allowed), and `optimizer` is one of the following letters:

o	default
p	primal simplex
d	dual simplex
n	network with dual simplex cleanup
h	barrier with crossover
b	barrier without crossover

```
s     sifting
c     concurrent
```

For example, this command:

```
lpex2 example.mps d
```

reads the file `example.mps` and solves the problem with the dual simplex optimizer.

To illustrate the ease of reading a problem, the example uses the routine `CPXreadcopyprob()`. This routine detects the type of the file, reads the file, and copies the data into the ILOG CPLEX problem object that is created with a call to `CPXcreateprob()`. The user need not be concerned with the memory management of the data. Memory management is handled transparently by `CPXreadcopyprob()`.

After calling `CPXopenCPLEX()` and turning on the screen indicator by setting the `CPX_PARAM_SCRIND` parameter to `CPX_ON`, the example creates an empty problem object with a call to `CPXcreateprob()`. This call returns a pointer, `lp`, to the new problem object. Then the data is read in by the routine `CPXreadcopyprob()`. After the data is copied, the appropriate optimization routine is called, based on the command line argument.

After optimization, the status of the solution is determined by a call to `CPXgetstat()`. The cases of infeasibility or unboundedness in the model are handled in a simple fashion here; a more complex application program might treat these cases in more detail. With these two cases out of the way, the program then calls `CPXsolninfo()` to determine the nature of the solution. Once it has been determined that a solution in fact exists, then a call to `CPXgetobjval()` is made, to obtain the objective function value for this solution and report it.

Next, preparations are made to print the solution value and basis status of each individual variable, by allocating arrays of appropriate size; these sizes are determined by calls to the routines `CPXgetnumcols()` and `CPXgetnumrows()`. Note that a basis is not guaranteed to exist, depending on which optimizer was selected at run time, so some of these steps, including the call to `CPXgetbase()`, are dependent on the solution type returned by `CPXsolninfo()`.

The primal solution values of the variables are obtained by a call to `CPXgetx()`, and then these values (along with the basis statuses if available) are printed, in a loop, for each variable. After that, a call to `CPXgetdblquality()` provides a measure of the numerical roundoff error present in the solution, by obtaining the maximum amount by which any variable's lower or upper bound is violated.

After the `TERMINATE:` label, the data for the solution (`x`, `cstat`, and `rstat`) are freed. Then the problem object is freed by `CPXfreeprob()`. After the problem is freed, the ILOG CPLEX environment is freed by `CPXcloseCPLEX()`.

Complete Program

The complete program follows. You can also view it online in the file `lpex2.c`.

```
/*--------------------------------------------------------------------------*/
/*  File: examples/src/lpex2.c                                              */
/*  Version 8.1                                                             */
/*--------------------------------------------------------------------------*/
/*  Copyright (C) 1997-2002 by ILOG.                                        */
/*  All Rights Reserved.                                                    */
/*  Permission is expressly granted to use this example in the              */
/*  course of developing applications that use ILOG products.               */
/*--------------------------------------------------------------------------*/

/* lpex2.c - Reading in and optimizing a problem */

/* To run this example, command line arguments are required.
   i.e.,   lpex2   filename   method
   where
       filename is the name of the file, with .mps, .lp, or .sav extension
       method   is the optimization method
                   o          default
                   p          primal simplex
                   d          dual   simplex
                   n          network with dual simplex cleanup
                   h          barrier with crossover
                   b          barrier without crossover
                   s          sifting
                   c          concurrent
   Example:
       lpex2  example.mps  o
 */

/* Bring in the CPLEX function declarations and the C library
   header file stdio.h with the following single include. */

#include <ilcplex/cplex.h>

/* Bring in the declarations for the string and character functions
   and malloc */

#include <ctype.h>
#include <stdlib.h>
#include <string.h>

/* Include declarations for functions in this program */

static void
   free_and_null (char **ptr),
   usage         (char *progname);
```

```
int
main (int argc, char *argv[])
{
   /* Declare and allocate space for the variables and arrays where we will
      store the optimization results including the status, objective value,
      maximum bound violation, variable values, and basis. */

   int      solnstat, solnmethod, solntype;
   double   objval, maxviol;
   double   *x     = NULL;
   int      *cstat = NULL;
   int      *rstat = NULL;

   CPXENVptr     env = NULL;
   CPXLPptr      lp = NULL;
   int           status = 0;
   int           j;
   int           cur_numrows, cur_numcols;
   int           method;

   char          *basismsg;

   /* Check the command line arguments */

   if (( argc != 3 )                                ||
       ( strchr ("podhbnsc", argv[2][0]) == NULL )  ) {
      usage (argv[0]);
      goto TERMINATE;
   }

   /* Initialize the CPLEX environment */

   env = CPXopenCPLEX (&status);

   /* If an error occurs, the status value indicates the reason for
      failure.  A call to CPXgeterrorstring will produce the text of
      the error message.  Note that CPXopenCPLEX produces no output,
      so the only way to see the cause of the error is to use
      CPXgeterrorstring.  For other CPLEX routines, the errors will
      be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON.  */

   if ( env == NULL ) {
      char  errmsg[1024];
      fprintf (stderr, "Could not open CPLEX environment.\n");
      CPXgeterrorstring (env, status, errmsg);
      fprintf (stderr, "%s", errmsg);
      goto TERMINATE;
   }

   /* Turn on output to the screen */

   status = CPXsetintparam (env, CPX_PARAM_SCRIND, CPX_ON);
```

```
if ( status ) {
   fprintf (stderr,
            "Failure to turn on screen indicator, error %d.\n", status);
   goto TERMINATE;
}

/* Create the problem, using the filename as the problem name */

lp = CPXcreateprob (env, &status, argv[1]);

/* A returned pointer of NULL may mean that not enough memory
   was available or there was some other problem.  In the case of
   failure, an error message will have been written to the error
   channel from inside CPLEX.  In this example, the setting of
   the parameter CPX_PARAM_SCRIND causes the error message to
   appear on stdout.  Note that most CPLEX routines return
   an error code to indicate the reason for failure.   */

if ( lp == NULL ) {
   fprintf (stderr, "Failed to create LP.\n");
   goto TERMINATE;
}

/* Now read the file, and copy the data into the created lp */

status = CPXreadcopyprob (env, lp, argv[1], NULL);
if ( status ) {
   fprintf (stderr, "Failed to read and copy the problem data.\n");
   goto TERMINATE;
}

/* Optimize the problem and obtain solution. */

switch (argv[2][0]) {
   case 'o':
      method = CPX_ALG_AUTOMATIC;
      break;
   case 'p':
      method = CPX_ALG_PRIMAL;
      break;
   case 'd':
      method = CPX_ALG_DUAL;
      break;
   case 'n':
      method = CPX_ALG_NET;
      break;
   case 'h':
      method = CPX_ALG_BARRIER;
      break;
   case 'b':
      method = CPX_ALG_BARRIER;
      status = CPXsetintparam (env, CPX_PARAM_BARCROSSALG, CPX_ALG_NONE);
      if ( status ) {
```

```
            fprintf (stderr,
                     "Failed to set the crossover method, error %d.\n",
status);
            goto TERMINATE;
         }
         break;
      case 's':
         method = CPX_ALG_SIFTING;
         break;
      case 'c':
         method = CPX_ALG_CONCURRENT;
         break;
      default:
         method = CPX_ALG_NONE;
         break;
   }

   status = CPXsetintparam (env, CPX_PARAM_LPMETHOD, method);
   if ( status ) {
      fprintf (stderr,
               "Failed to set the optimization method, error %d.\n", status);
      goto TERMINATE;
   }


   status = CPXlpopt (env, lp);
   if ( status ) {
      fprintf (stderr, "Failed to optimize LP.\n");
      goto TERMINATE;
   }

   solnstat = CPXgetstat (env, lp);

   if      ( solnstat == CPX_STAT_UNBOUNDED ) {
      printf ("Model is unbounded\n");
      goto TERMINATE;
   }
   else if ( solnstat == CPX_STAT_INFEASIBLE ) {
      printf ("Model is infeasible\n");
      goto TERMINATE;
   }
   else if ( solnstat == CPX_STAT_INFOrUNBD ) {
      printf ("Model is infeasible or unbounded\n");
      goto TERMINATE;
   }

   status = CPXsolninfo (env, lp, &solnmethod, &solntype, NULL, NULL);
   if ( status ) {
      fprintf (stderr, "Failed to obtain solution info.\n");
      goto TERMINATE;
   }
   printf ("Solution status %d, solution method %d\n", solnstat, solnmethod);
```

```
if ( solntype == CPX_NO_SOLN ) {
   fprintf (stderr, "Solution not available.\n");
   goto TERMINATE;
}

status = CPXgetobjval (env, lp, &objval);
if ( status ) {
   fprintf (stderr, "Failed to obtain objective value.\n");
   goto TERMINATE;
}
printf ("Objective value %.10g.\n", objval);

/* The size of the problem should be obtained by asking CPLEX what
   the actual size is.  cur_numrows and cur_numcols store the
   current number of rows and columns, respectively.  */

cur_numcols = CPXgetnumcols (env, lp);
cur_numrows = CPXgetnumrows (env, lp);


/* Retrieve basis, if one is available */

if ( solntype == CPX_BASIC_SOLN ) {
   cstat = (int *) malloc (cur_numcols*sizeof(int));
   rstat = (int *) malloc (cur_numrows*sizeof(int));
   if ( cstat == NULL || rstat == NULL ) {
      fprintf (stderr, "No memory for basis statuses.\n");
      goto TERMINATE;
   }

   status = CPXgetbase (env, lp, cstat, rstat);
   if ( status ) {
      fprintf (stderr, "Failed to get basis; error %d.\n", status);
      goto TERMINATE;
   }
}
else {
   printf ("No basis available\n");
}


/* Retrieve solution vector */

x = (double *) malloc (cur_numcols*sizeof(double));
if ( x == NULL ) {
   fprintf (stderr, "No memory for solution.\n");
   goto TERMINATE;
}

status = CPXgetx (env, lp, x, 0, cur_numcols-1);
if ( status ) {
   fprintf (stderr, "Failed to obtain primal solution.\n");
   goto TERMINATE;
```

```
   }


   /* Write out the solution */

   for (j = 0; j < cur_numcols; j++) {
      printf ( "Column %d:  Value = %17.10g", j, x[j]);
      if ( cstat != NULL ) {
         switch (cstat[j]) {
            case CPX_AT_LOWER:
               basismsg = "Nonbasic at lower bound";
               break;
            case CPX_BASIC:
               basismsg = "Basic";
               break;
            case CPX_AT_UPPER:
               basismsg = "Nonbasic at upper bound";
               break;
            case CPX_FREE_SUPER:
               basismsg = "Superbasic, or free variable at zero";
               break;
            default:
               basismsg = "Bad basis status";
               break;
         }
         printf ("  %s",basismsg);
      }
      printf ("\n");
   }

   /* Display the maximum bound violation. */

   status = CPXgetdblquality (env, lp, &maxviol, CPX_MAX_PRIMAL_INFEAS);
   if ( status ) {
      fprintf (stderr, "Failed to obtain bound violation.\n");
      goto TERMINATE;
   }
   printf ("Maximum bound violation = %17.10g\n", maxviol);


TERMINATE:

   /* Free up the basis and solution */

   free_and_null ((char **) &cstat);
   free_and_null ((char **) &rstat);
   free_and_null ((char **) &x);

   /* Free up the problem, if necessary */

   if ( lp != NULL ) {
      status = CPXfreeprob (env, &lp);
```

```
      if ( status ) {
         fprintf (stderr, "CPXfreeprob failed, error code %d.\n", status);
      }
   }

   /* Free up the CPLEX environment, if necessary */

   if ( env != NULL ) {
      status = CPXcloseCPLEX (&env);

      /* Note that CPXcloseCPLEX produces no output,
         so the only way to see the cause of the error is to use
         CPXgeterrorstring.  For other CPLEX routines, the errors will
         be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON. */

      if ( status ) {
         char  errmsg[1024];
         fprintf (stderr, "Could not close CPLEX environment.\n");
         CPXgeterrorstring (env, status, errmsg);
         fprintf (stderr, "%s", errmsg);
      }
   }

   return (status);

}  /* END main */



/* This simple routine frees up the pointer *ptr, and sets *ptr to NULL */

static void
free_and_null (char **ptr)
{
   if ( *ptr != NULL ) {
      free (*ptr);
      *ptr = NULL;
   }
} /* END free_and_null */


static void
usage (char *progname)
{
   fprintf (stderr,"Usage: %s filename algorithm\n", progname);
   fprintf (stderr,"   where filename is a file with extension \n");
   fprintf (stderr,"      MPS, SAV, or LP (lower case is allowed)\n");
   fprintf (stderr,"   and algorithm is one of the letters\n");
   fprintf (stderr,"      o          default\n");
   fprintf (stderr,"      p          primal simplex\n");
   fprintf (stderr,"      d          dual simplex\n");
   fprintf (stderr,"      n          network simplex\n");
```

```
   fprintf (stderr,"      b          barrier\n");
   fprintf (stderr,"      h          barrier with crossover\n");
   fprintf (stderr,"      s          sifting\n");
   fprintf (stderr,"      c          concurrent\n");
   fprintf (stderr," Exiting...\n");
} /* END usage */
```

Adding Rows to a Problem: Example `lpex3.c`

This example illustrates how to develop your own solution algorithms with routines from the Callable Library. It also shows you how to add rows to a problem object. Here is the problem example `lpex3` solves:

Minimize $c*x$

subject to $Hx = d$

$Ax = b$

$1 \leq x \leq u$

where

$$H = \begin{pmatrix} -1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & -1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & -1 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & 0 & -1 & 0 & 1 & -1 \end{pmatrix} \quad d = \begin{pmatrix} -3 \\ 1 \\ 4 \\ 3 \\ -5 \end{pmatrix}$$

$$A = \begin{pmatrix} 2 & 1 & -2 & -1 & 2 & -1 & -2 & -3 \\ 1 & -3 & 2 & 3 & -1 & 2 & 1 & 1 \end{pmatrix} \quad b = \begin{pmatrix} 4 \\ -2 \end{pmatrix}$$

$$c = \begin{pmatrix} -9 & 1 & 4 & 2 & -8 & 2 & 8 & 12 \end{pmatrix}$$

$$l = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$u = \begin{pmatrix} 50 & 50 & 50 & 50 & 50 & 50 & 50 & 50 \end{pmatrix}$$

The constraints $Hx=d$ represent a pure network flow. The example solves this problem in two steps:

1. The ILOG CPLEX Network Optimizer is used to solve

Minimize $c*x$

subject to $Hx = d$

$\boldsymbol{l} \leq x \leq u$

2. The constraints $Ax=b$ are added to the problem, and the dual simplex optimizer is used to solve the new problem, starting at the optimal basis of the simpler network problem.

The data for this problem consists of the network portion (using variable names beginning with the letter *H*) and the complicating constraints (using variable names beginning with the letter *A*).

The example first calls `CPXopenCPLEX()` to create the environment and then turns on the ILOG CPLEX screen indicator (`CPX_PARAM_SCRIND`). Next it sets the simplex display level (`CPX_PARAM_SIMDISPLAY`) to 2 to indicate iteration-by-iteration output, so that the progress of each iteration of the hybrid optimizer can be observed. Setting this parameter to 2 is not generally recommended; the example does so only for illustrative purposes.

The example creates a problem object by a call to `CPXcreateprob()`. Then the network data is copied via a call to `CPXcopylp()`. After the network data is copied, the parameter `CPX_PARAM_LPMETHOD` is set to `CPX_ALG_NET` and the routine `CPXlpopt()` is called to solve the network part of the optimization problem using the network optimizer. The objective value of this problem is retrieved by `CPXgetobjval()`.

Then the extra rows are added by calling `CPXaddrows()`. For convenience, the total number of nonzeros in the rows being added is stored in an extra element of the array `rmatbeg`, and this element is passed for the parameter `nzcnt`. The name arguments to `CPXaddrows()` are `NULL`, since no variable or constraint names were defined for this problem.

After the `CPXaddrows()` call, parameter `CPX_PARAM_LPMETHOD` is set to `CPX_ALG_DUAL` and the routine `CPXlpopt()` is called to re-optimize the problem using the dual simplex optimizer. After re-optimization, `CPXsolution()` is called to determine the solution status, the objective value, and the primal solution. `NULL` is passed for the other solution values, since they are not printed by this example.

At the end, the problem is written as a SAV file by `CPXwriteprob()`. This file can then be read into the ILOG CPLEX Interactive Optimizer to analyze whether the problem was correctly generated. Using a SAV file is recommended over MPS and LP files, as SAV files preserve the full numeric precision of the problem.

After the `TERMINATE:` label, `CPXfreeprob()` releases the problem object, and `CPXcloseCPLEX()` releases the ILOG CPLEX environment.

Complete Program

The complete program follows. You can also view it online in the file `lpex3.c`.

```
/*--------------------------------------------------------------------------*/
/*  File: examples/src/lpex3.c                                              */
/*  Version 8.1                                                             */
/*--------------------------------------------------------------------------*/
/*  Copyright (C) 1997-2002 by ILOG.                                        */
/*  All Rights Reserved.                                                    */
```

```
/*  Permission is expressly granted to use this example in the           */
/*  course of developing applications that use ILOG products.            */
/*---------------------------------------------------------------------*/

/* lpex3.c, example of using CPXaddrows to solve a problem */

/* Bring in the CPLEX function declarations and the C library
   header file stdio.h with the following single include. */

#include <ilcplex/cplex.h>

/* Bring in the declarations for the string functions */

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <math.h>
#endif

/* Modified example from Chvatal, "Linear Programming", Chapter 26.
 *   minimize  c*x
 *   subject to  Hx = d
 *               Ax = b
 *               l <= x <= u
 *   where
 *
 *   H = ( -1  0  1  0  1  0  0  0 )  d = ( -3 )
 *       (  1 -1  0  1  0  0  0  0 )      (  1 )
 *       (  0  1 -1  0  0  1 -1  0 )      (  4 )
 *       (  0  0  0 -1  0 -1  0  1 )      (  3 )
 *       (  0  0  0  0 -1  0  1 -1 )      ( -5 )
 *
 *   A = (  2  1 -2 -1  2 -1 -2 -3 )  b = (  4 )
 *       (  1 -3  2  3 -1  2  1  1 )      ( -2 )
 *
 *   c = ( -9  1  4  2 -8  2  8 12 )
 *   l = (  0  0  0  0  0  0  0  0 )
 *   u = ( 50 50 50 50 50 50 50 50 )
 *
 *
 *
 *  Treat the constraints with A as the complicating constraints, and
 *  the constraints with H as the "simple" problem.
 *
 *  The idea is to solve the simple problem first, and then add the
 *  constraints for the complicating constraints, and solve with dual.
 *
 */


#define  COLSORIG  8
#define  ROWSSUB   5
```

```
#define  NZSUB     (2*COLSORIG)
#define  ROWSCOMP  2
#define  NZCOMP    (ROWSCOMP*COLSORIG)
#define  ROWSTOT   (ROWSSUB+ROWSCOMP)
#define  NZTOT     (NZCOMP+NZSUB)


int
main()
{
   /* Data for original problem.  Arrays have to be big enough to hold
      problem plus additional constraints.  */

   double  Hrhs[ROWSTOT]     = { -3, 1, 4, 3, -5};
   double  Hlb[COLSORIG]     = { 0, 0, 0, 0, 0, 0, 0, 0};
   double  Hub[COLSORIG]     = { 50, 50, 50, 50, 50, 50, 50, 50 };
   double  Hcost[COLSORIG]   = { -9,  1,  4, 2, -8,  2,  8,  12 };
   char    Hsense[ROWSTOT]   = { 'E', 'E', 'E', 'E', 'E' };
   int     Hmatbeg[COLSORIG] = { 0, 2, 4, 6, 8, 10, 12, 14};
   int     Hmatcnt[COLSORIG] = { 2, 2, 2, 2, 2, 2, 2, 2};
   int     Hmatind[NZTOT]    = { 0, 1, 1, 2, 0, 2, 1, 3,
                                 0, 4, 2, 3, 2, 4, 3, 4};
   double  Hmatval[NZTOT]    = { -1.0, 1.0, -1.0, 1.0, 1.0, -1.0, 1.0, -1.0,
                                 1.0, -1.0, 1.0, -1.0, -1.0, 1.0, 1.0, -1.0 };

   /* Data for CPXaddrows call */

   double  Arhs[ROWSCOMP]    = { 4, -2};
   char    Asense[ROWSCOMP]  = { 'E', 'E' };
   /* Note - use a trick for rmatbeg by putting the total nonzero count in
             the last element.  This is not required by the CPXaddrows call. */
   int     Armatbeg[ROWSCOMP+1] = { 0, 8, 16};
   int     Armatind[NZCOMP]   = { 0, 1, 2, 3, 4, 5, 6, 7,
                                  0, 1, 2, 3, 4, 5, 6, 7  };
   double  Armatval[NZCOMP]   = {  2.0,  1.0, -2.0, -1.0,
                                   2.0, -1.0, -2.0, -3.0,
                                   1.0, -3.0,  2.0,  3.0,
                                  -1.0,  2.0,  1.0,  1.0 };

   double        x[COLSORIG];

   CPXENVptr     env = NULL;
   CPXLPptr      lp = NULL;

   int           j;
   int           status, lpstat;
   double        objval;

   /* Initialize the CPLEX environment */

   env = CPXopenCPLEX (&status);

   /* If an error occurs, the status value indicates the reason for
```

```
   failure.  A call to CPXgeterrorstring will produce the text of
   the error message.  Note that CPXopenCPLEX produces no output,
   so the only way to see the cause of the error is to use
   CPXgeterrorstring.  For other CPLEX routines, the errors will
   be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON.  */

if ( env == NULL ) {
   char  errmsg[1024];
   fprintf (stderr, "Could not open CPLEX environment.\n");
   CPXgeterrorstring (env, status, errmsg);
   fprintf (stderr, "%s", errmsg);
   goto TERMINATE;
}

/* Turn on output to the screen */

status = CPXsetintparam (env, CPX_PARAM_SCRIND, CPX_ON);
if ( status ) {
   fprintf (stderr,
            "Failure to turn on screen indicator, error %d.\n", status);
   goto TERMINATE;
}

status = CPXsetintparam (env, CPX_PARAM_SIMDISPLAY, 2);
if ( status ) {
   fprintf (stderr,"Failed to turn up simplex display level.\n");
   goto TERMINATE;
}

/* Create the problem */

lp = CPXcreateprob (env, &status, "chvatal");

if ( lp == NULL ) {
   fprintf (stderr,"Failed to create subproblem\n");
   status = 1;
   goto TERMINATE;
}

/* Copy network part of problem.  */

status = CPXcopylp (env, lp, COLSORIG, ROWSSUB, CPX_MIN, Hcost, Hrhs,
                    Hsense, Hmatbeg, Hmatcnt, Hmatind, Hmatval,
                    Hlb, Hub, NULL);

if ( status ) {
   fprintf (stderr, "CPXcopylp failed.\n");
   goto TERMINATE;
}


status = CPXsetintparam (env, CPX_PARAM_LPMETHOD, CPX_ALG_NET);
```

```
if ( status ) {
   fprintf (stderr,
            "Failed to set the optimization method, error %d.\n", status);
   goto TERMINATE;
}

status = CPXlpopt (env, lp);
if ( status ) {
   fprintf (stderr, "Failed to optimize LP.\n");
   goto TERMINATE;
}

status = CPXgetobjval (env, lp, &objval);
if ( status ) {
   fprintf (stderr,"CPXgetobjval failed\n");
   goto TERMINATE;
}

printf ("After network optimization, objective is %.10g\n", objval);

/* Now add the extra rows to the problem.  */

status = CPXaddrows (env, lp, 0, ROWSCOMP, Armatbeg[ROWSCOMP],
                     Arhs, Asense, Armatbeg, Armatind, Armatval,
                     NULL, NULL);
if ( status ) {
   fprintf (stderr,"CPXaddrows failed.\n");
   goto TERMINATE;
}

/* Because the problem is dual feasible with the rows added, using
   the dual simplex method is indicated.  */

status = CPXsetintparam (env, CPX_PARAM_LPMETHOD, CPX_ALG_DUAL);
if ( status ) {
   fprintf (stderr,
            "Failed to set the optimization method, error %d.\n", status);
   goto TERMINATE;
}

status = CPXlpopt (env, lp);
if ( status ) {
   fprintf (stderr, "Failed to optimize LP.\n");
   goto TERMINATE;
}

status = CPXsolution (env, lp, &lpstat, &objval, x, NULL, NULL, NULL);
if ( status ) {
   fprintf (stderr,"CPXsolution failed.\n");
   goto TERMINATE;
}

printf ("Solution status %d\n",lpstat);
```

```
   printf ("Objective value %g\n",objval);
   printf ("Solution is:\n");
   for (j = 0; j < COLSORIG; j++) {
      printf ("x[%d] = %g\n",j,x[j]);
   }

   /* Put the problem and basis into a SAV file to use it in the
    * Interactive Optimizer and see if problem is correct */

   status = CPXwriteprob (env, lp, "lpex3.sav", NULL);
   if ( status ) {
      fprintf (stderr, "CPXwriteprob failed.\n");
      goto TERMINATE;
   }


TERMINATE:

   /* Free up the problem as allocated by CPXcreateprob, if necessary */

   if ( lp != NULL ) {
      status = CPXfreeprob (env, &lp);
      if ( status ) {
         fprintf (stderr, "CPXfreeprob failed, error code %d.\n", status);
      }
   }

   /* Free up the CPLEX environment, if necessary */

   if ( env != NULL ) {
      status = CPXcloseCPLEX (&env);

      /* Note that CPXcloseCPLEX produces no output,
         so the only way to see the cause of the error is to use
         CPXgeterrorstring.  For other CPLEX routines, the errors will
         be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON. */

      if ( status ) {
         char  errmsg[1024];
         fprintf (stderr, "Could not close CPLEX environment.\n");
         CPXgeterrorstring (env, status, errmsg);
         fprintf (stderr, "%s", errmsg);
      }
   }

   return (status);

} /* END main */
```

Performing Sensitivity Analysis

In *Performing Sensitivity Analysis* on page 46, there is a discussion of how to perform sensitivity analysis in the Interactive Optimizer. As with most interactive features of ILOG CPLEX, there is a direct approach to this task from the Callable Library. Here we modify the example `lpex1.c` on page 111 to show how to perform sensitivity analysis with routines from the Callable Library.

We suggest that you make a copy of `lpex1.c`, and edit this new source file. Among the declarations (for example, immediately after the declaration for `dj`) insert these additional declarations:

```
double *lowerc = NULL, *upperc = NULL;
double *lowerr = NULL, *upperr = NULL;
```

At some point after the call to `CPXlpopt()`, (for example, just before the call to `CPXwriteprob()`), perform sensitivity analysis on the objective function and on the right-hand side coefficients by inserting this fragment of code:

```
upperc = (double *) malloc (cur_numcols * sizeof(double));
lowerc = (double *) malloc (cur_numcols * sizeof(double));
status = CPXobjsa (env, lp, 0, cur_numcols-1, lowerc, upperc);
if ( status ) {
     fprintf (stderr, "Failed to obtain objective sensitivity.\n");
     goto TERMINATE;
}
printf ("\nObjective coefficient sensitivity:\n");
for (j = 0; j < cur_numcols; j++) {
  printf ("Column %d:  Lower = %10g  Upper = %10g\n",
          j, lowerc[j], upperc[j]);
}

upperr = (double *) malloc (cur_numrows * sizeof(double));
lowerr = (double *) malloc (cur_numrows * sizeof(double));
status = CPXrhssa (env, lp, 0, cur_numrows-1, lowerr, upperr);
if ( status ) {
  fprintf (stderr, "Failed to obtain RHS sensitivity.\n");
  goto TERMINATE;
}
printf ("\nRight-hand side coefficient sensitivity:\n");
for (i = 0; i < cur_numrows; i++) {
  printf ("Row %d:  Lower = %10g  Upper = %10g\n",
          i, lowerr[i], upperr[i]);
}
```

This sample is familiarly known as “throw away” code. For production purposes, you probably want to observe good programming practices such as freeing these allocated arrays at the `TERMINATE` label in the application.

A bound value of $1e^{+20}$ (`CPX_INFBOUND`) is treated as infinity within ILOG CPLEX, so this is the value printed by our sample code in cases where the upper or lower sensitivity range

on a row or column is infinite; a more sophisticated program might print a string, such as `-inf` or `+inf`, when negative or positive `CPX_INFBOUND` is encountered as a value.

Similar code could be added to perform sensitivity analysis with respect to bounds via `CPXboundsa()`.

Index

A

accessing
  basic rows and columns of solution **45**
  basis information **81**
  dual values **45**
  objective function value **45**
  reduced costs **45**
  slack values **45**
  solution values **45**, **70**
`add` Interactive Optimizer command **55**
  syntax **56**
`add(obj)` **94**
adding
  bounds **55**
  constraint to model **86**
  constraints **55**
  from a file **56**
  interactively **55**
  objective function to model **69**
  rows to a problem **131**
`addLe` **97**
`addMinimize()` **97**
`addMinimize(expr)` **94**
advanced basis
  advanced start indicator **44**
  using **50**
algorithm
  automatic (`AutoAlg`) **80**
  creating object **69**, **72**
`and()` **98**
application
  and Callable Library **13**
  and Concert Technology **13**
  compiling and linking Callable Library **104**
  compiling and linking Component Libraries **29**
  compiling and linking Concert Technology **65**
  development steps **108**
  error handling **71**, **110**

B

`baropt` Interactive Optimizer command **44**
barrier optimizer
  availability **44**
  selecting **80**
BAS file format **50**, **53**
basis
  accessing information **81**
  basis information **95**
  periodically written **50**
  starting from previous **87**
basis file
  reading **53**
  writing **50**
BIN file format **48**
boolean parameter **86**
boolean variable
  representing in model **68**
bound

adding **55**
changing **58**
default values **36**
displaying **42**
entering in LP format **36**
removing **58**
sensitivity analysis **47**, **139**
box variable **39**
branch & bound **80**
branch & cut **80**

C

Callable Library **103** to **139**
application development steps **108**
compiling and linking applications **104**
conceptual design **103**
CPLEX operation **106**
description **13**
distribution file **104**
error handling **110**
example model **18**
opening CPLEX **106**
see also individual CPX*xxx* routines
`change` Interactive Optimizer command **56**
`bounds` **58**
change options **56**
`coefficient` **59**
`delete` **59**
delete options **59**
`objective` **59**
`rhs` **59**
`sense` **57**
syntax **60**
changing
bounds **58**
coefficients **58**
constraint names **57**
parameters **54**, **86**
problem **56**
sense **57**
variable names **57**
choosing
optimizer **44**, **80**, **110**
class library **90**
classpath **91**
command line option **90**
coefficient
changing **58**
column
expressions **73**
command
executing from operating system **60**
input formats **32**
Interactive Optimizer list **33**
compiler
-DNDEBUG option **71**
error messages **66**
Microsoft Visual C++ Command Line **105**
using with CPLEX **65**
compiling
applications **29**
Callable Library applications **104**
Concert Technology applications **65**
Component Libraries
defined **13**
running examples **28**
verifying installation **28**
Concert Technology Library **63** to **88**
C++ classes **66**
C++ objects **64**
compiling and linking applications **65**
CPLEX design in **64**
description **13**
error handling **71**
example model **16**
running examples **65**
see also individual Ilo*xxx* routines
constraint
adding **55**, **86**
changing names **57**
changing sense **57**
creating **73**
default names **36**
deleting **59**
displaying **41**
displaying names **40**
displaying nonzero coefficients **39**
displaying number of **39**
displaying type **39**

entering in LP format **36**
name limitations **36**
naming **36**
range **73**
representing in model **68**
constraints
adding to a model **94**
continuous variable
representing **68**
cost
reduced **95**
CPLEX
compatible platforms **13**
Component Libraries **13**
description **12**
directory structure **27**
installing **26**
licensing **28**
problem types **12**
quitting **61**
setting up **25**
starting **32**
technologies **13**
Web site **23**
`cplex`
solve the model **94**
`cplex` command **32**
cplex.jar
location **89**
`cplex.log` file **44**
`cplex.numVarArray()` **97**
`cplex.out` **96**
`cplex.setOut` **96**
`cplex.setWarning()` **96**
`cplex.solve()` **96**
`cplex.warning` **96**

CPX

`CPXaddcols` routine **107**, **109**, **112**
`CPXaddrows` routine **107**, **109**, **112**, **132**
`CPXboundsa` routine **139**
`CPXchgcoeflist` routine **107**, **109**, **112**
`CPXcloseCPLEX` routine **107**, **113**, **123**, **132**
`CPXcopylp` routine **107**, **108**, **109**, **113**, **132**
`CPXcreateprob` routine **107**, **123**, **132**
`CPXfreeprob` routine **107**, **113**, **123**, **132**
`CPXgeterrorstring` routine **112**, **113**
`CPXgetobjval` routine **132**
`CPXlpopt` routine **112**, **132**, **138**
`CPXmsg` routine **106**
`CPXnewcols` routine **107**, **109**, **112**
`CPXnewrows` routine **107**, **109**, **112**
`CPXopenCPLEX` routine **106**, **112**, **123**, **132**
`CPXreadcopyprob` routine **107**, **123**
`CPXsetintparam` routine **112**
`CPXsolution` routine **112**, **132**
`CPXwriteprob` routine **111**, **113**, **132**, **138**

C (continued)

create
model **93**
creating
algorithm object **69**, **72**
automatic log file **44**
binary problem representation **111**
constraint **73**
environment **132**
environment object **66**, **72**
model (`IloModel`) **67**
model object **72**
objective function **73**, **79**
optimization model **67**, **68**
problem files **47**
problem object **107**, **132**
solution files **48**
SOS **79**
variable **79**

D

data
entering **37**
entry options **14**
deleting
constraints **59**
problem options **59**
variables **59**
directory

installation structure **27**
`display` Interactive Optimizer command **38**, **57**
display options **38**
`problem` **38**
`bounds` **42**
`constraints` **41**
display problem options **38**
`names` **40**, **41**
`stats` **39**
syntax **39**
`sensitivity` **46**
syntax **47**
`settings` **55**
`solution` **45**
syntax **46**
specifying item ranges **40**
syntax **42**
displaying
basic rows and columns **45**
bounds **42**
constraint names **40**
constraints **41**
nonzero constraint coefficients **39**
number of constraints **39**
objective function **41**
optimal solution **43**
parameter settings **55**
post-solution information **45**
problem **38**
problem options **38**
problem part **39**
problem statistics **39**
sensitivity analysis **46**, **138**
type of constraint **39**
variable names **40**
variables **39**
DPE file format **48**
DUA file format **48**
dual simplex optimizer
as default **42**
availability **44**
finding a solution **112**
selecting **80**
dual values
accessing **45**
duals **95**

E

EMB file format **48**
`enter` Interactive Optimizer command **34**
`bounds` **36**
`maximize` **35**
`minimize` **35**
`subject to` **36**, **55**
syntax **35**
entering
bounds **36**
constraint names **36**
constraints **36**
example problem **34**
item ranges **40**
keyboard data **37**
objective function **35**, **36**
objective function names **36**
problem **34**, **35**
problem name **34**
variable bounds **36**
variable names **35**
Environment
construct **66**
environment object
creating **66**, **72**
destroying **67**
memory management and **67**
equality constraints
add to a model **94**
Error
NoClassDefFoundError **91**
error
invalid encrypted key **91**
no license found **91**
UnsatisfiedLinkError **91**
error handling
programming errors **71**
runtime errors **71**
testing installation **29**, **66**
error message
compiler **66**
license manager **66**

linker **66**
example
adding rows to a problem **131**
entering a problem **34**
entering and optimizing a problem **111**
ilolpex2.cpp **80**
ilolpex3.cpp **85**
lpex1.c **111**
lpex2.c **122**
lpex3.c **131**
modifying an optimization problem **85**
reading a problem file **122**
reading a problem from a file **80**
running Callable Library **105**
running Component Libraries **28**
running Concert Technology **65**
running from standard distribution **105**
solving a problem **42**
exception
handling **71**
executing
operating system commands **60**
exportModel member function
IloCplex class **79**
expression
column **73**

F

False **94**
feasible solution **94**
file format
read options **51**
write options **48**
file name
extension **49**, **52**, **79**

G

getCplexStatus **95**
getCplexStatus member function
IloCplex class **70**
getDuals member function
IloCplex class **73**
getObjValue member function
IloCplex class **70**
getReducedCosts member function
IloCplex class **73**
getSlacks member function
IloCplex class **73**
getStatus **94**
getStatus member function
IloCplex class **70**, **73**
getValue member function
IloCplex class **70**
getValues member function
IloCplex class **73**
greater than equal to constraints
add to a model **94**

H

handle class
definition **67**
empty handle **68**
handling
errors **71**, **110**
exceptions **71**
help Interactive Optimizer command **32**
syntax **33**

I

IIS file format **48**

IloA

IloAddNumVar class **73**
IloAlgorithm::Exception class **71**
IloAlgorithm::Status enumeration **73**

IloC

IloColumn.and() **98**
IloCplex
add modeling object **94**
class **89**
IloCplex class **64**, **69**
exportModel member function **79**
getCplexStatus member function **70**

getDuals member function **73**
getObjValue member function **70**
getReducedCosts member function **73**
getSlacks member function **73**
getStatus member function **70**, **73**
getValue member function **70**
getValues member function **73**
importModel member function **79**, **81**
setParam member **80**
setRootAlgorithm member function **81**
solve member function **70**, **73**, **81**, **85**
IloCplex.addLe **97**
IloCplex.addMinimize() **97**
IloCplex.prod() **97**
IloCplex.scalProd() **97**
IloCplex.sum() **97**
IloCplex::Algorithm enumeration **80**
IloCplex::BoolParam enumeration **86**
IloCplex::Exception class **71**
IloCplex::IntParam enumeration **86**
IloCplex::NumParam enumeration **86**
IloCplex::StringParam enumeration **86**

IloE

IloEnv class **66**
end member function **67**
IloException class **71**
IloExpr class **69**
IloExtractable class **67**

IloG

ILOG
technical support **23**
Web sites **23**
ILOG License Manager (ILM)
CPLEX and **28**
ILOG_LICENSE_FILE environment variable **28**

IloL

IloLinearNumExpr **93**

IloM

IloMinimize function **69**
IloModel class **64**, **68**
add member function **68**, **69**

IloN

IloNumArray class **73**
IloNumColumn class **73**
IloNumExpr **93**
IloNumVar **93**
IloNumVar class **68**, **74**, **79**

IloO

IloObjective **93**
IloObjective class **68**, **73**, **79**
setCoef member function **74**

IloR

IloRange **93**
IloRange class **68**, **69**, **73**, **79**
setCoef member function **74**
IloRange.setExpr() **98**

IloS

IloSemiContVar class **79**
IloSOS1 class **79**
IloSOS2 class **79**

I (continued)

importModel member function
IloCplex class **79**, **81**
infeasible **95**
installing CPLEX **25** to **29**
testing installation **28**
integer parameter **86**
integer variable
optimizer used **110**
representing in model **68**
Interactive Optimizer **31** to **61**

command formats **32**
commands **33**
description **13**
example model **15**
quitting **61**
starting **32**
invalid encrypted key **91**
iteration log **43**, **44**

J

Java Native Interface **89**
Java Virtual Machine **90**
javamake **90**
JNI **89**
JVM **90**

L

libformat **90**
licensing
CPLEX **28**
linear optimization **12**
link
Concert Technology library files **29**
CPLEX library files **29**
linker
error messages **66**
using with CPLEX **65**
linking
applications **29**
Callable Library applications **104**
Concert Technology applications **65**
log file
adding to **54**
`cplex.log` **44**
creating **44**
iteration log **43**, **44**
LP
creating a model **15**
node **80**
problem format **12**
root **80**
solving a model **15**
solving pure **80**
LP file
reading **51**
writing **49**
LP file format **35**
lpex1.c **138**
`LPex1.java` **95**
LPMETHOD parameter **42**

M

makefile **90**
maximization
in LP problem **35**
memory management
by environment object **67**
MIN file format **48**
minimization
in LP problem **35**
MIP
description **12**
solving **80**
MIP optimizer
availability **44**
`mipopt` Interactive Optimizer command **44**
Model
creating **67**
model
adding constraints **86**
creating `IloModel` **67**
extracting **72**
modifying **85**
reading from file **79**, **81**
solving **81**
writing to file **79**
model object
creating **72**
`model.column()` **98**
modeling
by column **73**
by nonzerows **74**
objects **64**
`modeling`
by columns **97**
Modeling by Columns **72**
modeling by columns **95**

Modeling by Nonzero Elements **72**
modeling by nonzeros **95**, **98**
Modeling by Rows **72**
modeling by rows **73**, **95**, **97**
modeling variables **93**
modifying
  problem object **107**
monitoring
  iteration log **43**
MPS file format **52**
MST file format **48**
multiple algorithms
  selecting **80**

N

NET file format **49**
`netopt` Interactive Optimizer command **44**
network
  description **12**
  flow **86**
network optimizer
  availability **44**
  selecting **80**
  solving with **86**
Nmake **90**
no license found **91**
NoClassDefFoundError **91**
node LP
  solving **80**
nonzeros
  modeling by **74**
notation in this manual **21**
notification **85**
numeric parameter **86**
`numVar()` **98**
`numVarArray()` **97**

O

objective function
  accessing value **45**
  adding to model **69**
  changing coefficient **59**
  changing sense **58**
  creating **73**, **79**
  default name **36**
  displaying **41**
  entering **36**
  entering in LP format **35**
  name **36**
  representing in model **68**
  sensitivity analysis **46**, **138**
operator() **73**
operator+ **73**
optimal solution **94**
optimization
  interrupting **44**
optimization model
  creating **67**
  defining extractable objects **68**
  extracting **67**
optimization problem
  reading from file **80**
  representing **72**
  solving with `IloCplex` **69**
`optimize` Interactive Optimizer command **42**
  re-solving **44**
  syntax **43**
optimizer
  choosing **44**, **80**, **81**, **110**
  options **13**
  parallel **14**, **106**
ORD file format **49**
ordering
  variables **41**
`OutputStream` **96**

P

parallel
  optimizers **14**, **106**
parameter
  boolean **86**
  changing **54**, **86**
  displaying settings **55**
  integer **86**
  list of settable **54**
  numeric **86**
  resetting to defaults **54**

string **86**
parameter specification file **55**
path names
using **50**
`populateByColumn` **96**
`populateByNonzero` **96**
`populateByNonzero()` **98**
`populateByRow` **96**
PPE file format **48**
PRE file format **49**
primal simplex optimizer
availability **44**
selecting **80**
`primopt` Interactive Optimizer command **44**
problem
change options **57**
changing **56**
creating binary representation **111**
data entry options **14**
display options **38**
displaying **38**
displaying a part **39**
displaying statistics **39**
entering from the keyboard **34**
entering in LP format **35**
naming **34**
reading files **122**
solving **42**, **112**
verifying entry **38**, **57**
problem file
reading **51**
writing **47**
problem formulation
`lpex1.c` **112**
problem object
creating **107**
modifying **107**
problem type
solved by CPLEX **12**
`prod()` **97**

Q

QP
description **12**
solving pure **80**
QP file format **49**
QP model
applicable solution algorithms **80**
`quit` Interactive Optimizer command **61**
quitting
ILOG CPLEX **61**
Interactive Optimizer **61**

R

range constraint **73**
ranged constraints
add to a model **94**
`read` Interactive Optimizer command **51**, **52**, **53**
file type options **51**
syntax **53**
reading
file format for **51**
LP files **51**
model from file **79**, **81**
MPS files **52**
problem files **51**, **122**
reduced cost **95**
reduced costs
accessing **45**
removing bounds **58**
representing
optimization problem **72**
re-solving **44**
REW file format **49**
right-hand side (RHS)
changing coefficient **59**
sensitivity analysis **46**, **138**
root LP
solving **80**

S

SAV file format **49**, **132**
saving
problem files **47**
solution files **47**
`scalProd()` **97**
sense

changing **57**
sensitivity analysis
performing **46**, **138**
`set` Interactive Optimizer command **54**
`advance` **44**
available parameters **54**
`defaults` **54**
`logfile` **44**
`simplex` **43**
`basisinterval` **50**
syntax **55**
`setExpr()` **98**
`setRootAlgorithm` member function
`IloCplex` class **81**
setting
parameters **54**, **86**
parameters to default **54**
sifting
select algorithm **80**
slack
accessing values **45**
slacks **95**
solution
accessing basic rows and columns **45**
accessing values **45**
displaying **45**
displaying basic rows and columns **45**
outputting **73**
process **43**
querying results **70**
reporting optimal **43**
restarting **44**
sensitivity analysis **46**, **138**
solution file
writing **47**
`solve` **94**
`solve` member function
`IloCplex` class **70**, **73**, **81**, **85**
solving
model **69**, **81**
node LP **80**
problem **42**, **112**
root LP **80**
with network optimizer **86**
SOS
creating **79**
SOS file format **49**
sparse matrix **86**
starting
CPLEX **32**
from previous basis **87**
Interactive Optimizer **32**
new problem **34**
string parameter **86**
structure of a CPLEX application **92**
Supported Platforms **90**
`System.out` **96**

T

technical support **23**
`tranopt` Interactive Optimizer command **44**
TRE file format **49**
`True` **94**
TXT file format **48**

U

unbounded **94**
UNIX
building Callable Library applications **105**
executing commands **61**
installation directory **26**
installing CPLEX **26**
testing CPLEX in Concert Technology **65**
verifying installation **28**
UnsatisfiedLinkError **91**

V

variable
boolean **68**
box **39**
changing bounds **58**
changing names **57**
continuous **68**
creating **79**
deleting **59**
displaying **39**
displaying names **40**

entering bounds **36**
entering names **35**
integer **68**
name limitations **35**
ordering **41**
removing bounds **58**
representing in model **68**
variables
modeling **93**
VEC solution File format **49**

W

Web site **23**
Windows
building Callable Library applications **105**
dynamic loading **106**
installing CPLEX **26**
Microsoft Visual C++ compiler **105**
Microsoft Visual C++ IDE **105**
testing CPLEX in Concert Technology **65**
verifying installation **29**
`write` Interactive Optimizer command **47**, **48**, **49**
file type options **48**
syntax **50**
writing
basis files **50**
file format for **48**
LP files **49**
model to file **79**
problem files **47**
solution files **47**

X

`xecute` Interactive Optimizer command **60**
syntax **61**
xxx file format **50**