Getting Started with Using CPLEX in C++

Illinois Institute of Technology
Department of Applied Mathematics

Adam Rumpf
arumpf@hawk.iit.edu

December 18, 2016

Contents

1 Introduction

CPLEX is one of the most widely used solvers for linear programs and mixed-integer programs. It is used extensively in the operations research community, both in academia and industry. Knowing how to use it effectively is an invaluable skill for any applied mathematician.

The IBM ILOG CPLEX Optimization Studio distribution comes with an IDE for solving models, and it is available as a solver in several popular computer algebra systems, notably AMPL. There is also an API that allows it to be directly accessed through common programming languages, including Python, Java, C#, and C++.

Accessing CPLEX in this way offers several advantages: First of all, despite being easier to use, the free student version of AMPL places restrictions on its problem sizes (currently to 500 variables and 500 constraints). There is no such restriction when accessing CPLEX through the API. Second, it is often the case that solving a single LP or MILP is only part of what needs to be done in an application. For example, we might want to apply a metaheuristic that requires solving an LP in each iteration. Being able to call CPLEX from within a larger program allows for much more flexibility in modeling applications.

This guide is meant to explain how to get started using CPLEX in C++, including how to obtain the necessary programs and how to build a very simple model. It is by no means comprehensive. The best way to learn more about this process is by practicing with your own examples, and by consulting the official documentation.

1.1 Obtaining C++

As one of the most widely used programming languages in the world, there are a great many IDEs available for C++. For the purposes of this guide, we will be using Microsoft Visual Studio, which is available to students for free from `https://www.visualstudio.com/downloads/`.

This guide will assume basic familiarity with the syntax and usage of C++. If you are new to C++, or you simply need to review some topics, see `http://www.cplusplus.com/` for comprehensive guides and documentation.

1.2 Obtaining CPLEX

The CPLEX optimizer is available through several common computer algebra systems. To access it through the C++ API, however, we must download the standalone CPLEX package from IBM. It is available to IIT students for free through the IBM Student Developer Community.

To download your copy, do the following:

1. Visit the IBM Student Developer Community at `https://developer.ibm.com/students/`
2. At the bottom of the page, fill in the information for IIT.
3. Follow the link that appears to the **Illinois Institute of Technology WebStore**, or follow the link to **View all software**.
4. From the WebStore, navigate to **Students > IBM > Data & Analytics > CPLEX**.
5. Click the link to **IBM ILOG CPLEX Optimization Studio – Student**.
6. Add the program to your cart (it should be listed as free).
7. In order to check out and download the program, you will need to create an account with IBM developerWorks, which requires providing your IIT email address.

After downloading and installing the program, you will have access to the IBM ILOG CPLEX Studio IDE, a standalone program that can be used for building and solving LP and MILP models. The installation also includes the files needed to access the CPLEX optimizer through other programs, which will be explained in the next section.

2 Using CPLEX in C++

This section will outline the process for how to use CPLEX in a Visual Studio C++ project, from the compiler configuration to the code, itself. By this point both Visual Studio and CPLEX should be installed.

Special thanks to Daniel Simmons and Dr. Qipeng Phil Zheng[1], and to Dr. Leandro C. Coelho[2], whose guides proved invaluable in figuring out the linking process for my own installations.

2.1 Linking CPLEX to Visual Studio

The CPLEX Studio download includes several libraries of classes that allow a C++ program to build and solve models in CPLEX. In order to access these classes, we must first configure our compiler to read from these libraries. The instructions below outline this process for my particular installations, which are:

Microsoft Visual Studio Community 2015

CPLEX 12.7.0

On my system, these were both installed to their default directories. If you choose to install them to a different directory, some of the file paths shown below will need to be adjusted.

1. Open Visual Studio and start an empty project.

2. There are a few compiler settings built into the CPLEX libraries that need to be matched in our Visual Studio project. For example, if we are using the 64-bit version of the CPLEX libraries, but the compiler is set to 32-bit mode, we will run into errors when we try to compile the code.

   To fix this, from the main project window, navigate to **Build** > **Configuration Manager**. Make sure that **Active solution configuration** is set to **Release**, and that **Active solution platform** is set to whatever matches your CPLEX installation (for me, this is **x64**).

   Note that changing these settings resets any changes made in the Property Pages window, so be sure to complete this step *before* the rest.

3. Now we begin the process of actually linking CPLEX to Visual Studio. From the main project window, navigate to **Project** > **[project name] Properties** (or press [Alt]+[F7]). This opens the Property Pages window.

   From here, navigate to **Configuration Properties** > **C/C++** > **General**. Edit the **Additional Include Directories** field to include the file paths to the `include` folders in both the CPLEX and the CONCERT directories, separated by a semicolon. For me, these are:

   ```
   C:\Program Files\IBM\ILOG\CPLEX_Studio127\cplex\include;
   C:\Program Files\IBM\ILOG\CPLEX_Studio127\concert\include
   ```

4. Still under C/C++ Properties, navigate to **Preprocessor** and add the following to the **Preprocessor Definitions** field:

   ```
   WIN32;
   _CONSOLE;
   IL_STD
   ```

5. Still in the Property Pages window, navigate to **Linker** > **General**. Edit the **Additional Library Directories** field to include the `stat_mda` folders in both the CPLEX and the CONCERT directories, separated by a semicolon. For me, these are:

[1] http://www.iems.ucf.edu/qzheng/grpmbr/seminar/Daniel_Using_C++_with_CPLEX.pdf
[2] http://www.leandro-coelho.com/how-to-configure-ms-visual-studio-to-use-ibm-cplex-concert/

```
C:\Program Files\IBM\ILOG\CPLEX_Studio127\cplex\lib\x64_windows_vs2015\stat_mda;
C:\Program Files\IBM\ILOG\CPLEX_Studio127\concert\lib\x64_windows_vs2015\stat_mda
```

6. Still under Linker, navigate to **Input**. Edit the **Additional Dependencies** field to contain the `.lib` files for CPLEX (including your version number, which for me is 12.7.0), CONCERT, and ILOCPLEX, separated by semicolons. For me, these are:

```
cplex1270.lib;
concert.lib;
ilocplex.lib
```

7. Click **OK** to apply these settings and close the Property Pages window. You should now be ready to start using the CPLEX libraries in your C++ project.

2.2 Building a Model

In order to use CPLEX models in a C++ project, the following must be included in the preprocessor directives:

```
#include "ilcplex\cplex.h"
#include "ilcplex\ilocplex.h"
```

The CPLEX API for C++ takes advantage of the object-oriented nature of C++. The workflow for solving one or more optimization problems revolves primarily around three types of objects: environments, models, and Cplex objects (we will spell the C++ object as "Cplex" rather than "CPLEX" to avoid confusion between it and the CPLEX solver).

- **Environment** objects manage the memory for model objects and algorithms. For our purposes, simply think of them as containing the variable definitions (including each variables' name, upper and lower bounds, and data type). A single environment can be associated with several models that share the same set of variable definitions.
- **Model** objects contain all objects necessary to define a problem, including variables, constraints, and objectives. Each of these is its own type of object.
- **Cplex** objects extract information from model objects for use in an appropriate CPLEX solution algorithm. They contain methods for setting aspects of the CPLEX algorithm, calling CPLEX to obtain a solution, and exporting information about the solution.

All of the classes we will use begin with the prefix `Ilo`. Full documentation for all classes used in this section can be found through the IBM Knowledge Center[3]. Also see the official guide[4], which provides an extremely comprehensive explanation of how to get started (see Chapter 4 for a series of C++ tutorials).

The basic workflow for solving a single problem through CPLEX is as follows:

1. Define an environment object.
2. Define a model object in this environment.
3. Add variables to the environment.
4. Add constraints and an objective to the model.

[3] http://www.ibm.com/support/knowledgecenter/SSSA5P_12.5.0/kc_gen/ilog.odms.ide.help_toc-gen2.html
[4] https://www.ibm.com/support/knowledgecenter/SSSA5P_12.6.3/ilog.odms.studio.help/pdf/gscplex.pdf

5. Define a Cplex object.
6. Extract the model to CPLEX.
7. Solve the model.
8. Export any required information about the solution.
9. Close the environment.

We will now look at each major step in detail.

Defining an Environment and a Model

To create a new environment object, and an associated model object, use the following syntax:

```
IloEnv myenv;
IloModel mymodel(myenv);
```

`IloEnv` and `IloModel` are the classes for environments and models, respectively. The names `myenv` and `mymodel` can be replaced with any valid name.

Adding Variables to the Environment

If there are few enough variables in the problem to name each individually, the simplest way to add them to the environment is by using the `IloNumVar` class as follows:

```
IloNumVar myvar(env, lower, upper, type);
```

where

- `myvar` is the name of the variable.
- `env` is the environment object to which the variable belongs.
- `lower` and `upper` are numerical lower and upper bounds, respectively. If the variable is meant to be unbounded from above, use an upper bound of `IloInfinity`, and if unbounded from below, a lower bound of `-IloInfinity`.
- `type` can be specified as `ILOFLOAT`, `ILOINT`, or `ILOBOOL` (all caps) to specify that the variable is continuous, integer, or binary, respectively. The argument defaults to `ILOFLOAT` if excluded.

However, it is often the case that there are too many variables to reasonably define each individually. In this case, we can use the `IloNumVarArray` class as follows:

```
IloNumVarArray myvar(env, n, lower, upper, type);
```

This time `myvar` will be treated as an array of `n` variables, all with the same environment, bounds, and type. Specific variables from this array are called as they would be in a standard C++ array (i.e. the first element is `myvar[0]`, the second is `myvar[1]`, the third is `myvar[2]`, and so on until reaching the final element `myvar[n-1]`).

The bounds of a variable can be changed using the `IloNumVarArray::setBounds` method as follows:

```
myvar[i].setBounds(lower, upper);
```

The above code will change the bounds of element `i` of variable array `myvar` to the specified limits.

Adding Constraints to the Model

Constraints are added using the `IloModel::add` method as follows:

```
model.add(constraint);
```

Here, `constraint` can be either an `IloRange` object (if a single constraint) or an `IloRangeArray` object (if an array of constraints). For example, to enter an individual constraint, we can use the syntax:

```
model.add(IloRange(env, lower, expr, upper));
```

where

- `model` is the name of the model to which the constraint is being added.
- `env` is the environment to which `model` belongs.
- `lower`, `expr`, and `upper` define a constraint of the form `lower` $\leq$ `expr` $\leq$ `upper`, where `lower` and `upper` are constants and `expr` is a linear combination of variables. If `lower` and `upper` have the same value, then this represents an equality constraint. `lower` and `upper` can each be made infinite via `IloInfinity`. The `expr` part of the constraint is an `IloExpr` object, and can be typed using the standard C++ syntax for arithemtic (for example, `2*myvar[0] + myvar[1] - 4*myvar[2]`).

For example, suppose we have already defined an environment called `myenv`, a model called `mymodel`, and an array of variables of length 3 called `x`, and that we want to define a constraint of the form $2x_1 + x_2 - 4x_3 \geq 6$. The following code shows how to define this constraint.

```
mymodel.add(IloRange(myenv, 6, 2*x[0] + x[1] - 4*x[2], IloInfinity));
```

The following code achieves the same result.

```
mymodel.add(6 <= 2*x[0] + x[1] - 4*x[2]);
```

As with variables, it is often not possible to type each constraint individually. In this case, we can use loops in our program to generate the constraints. This method is useful if the constraints are being read in from an external file. There are many different ways of defining constraints automatically. The following method shows how to build the constraints one row at a time:

1. Begin a `for` loop with one iteration for each constraint.
2. Within each iteration of this loop, do the following:
   (a) Initialize an empty expression from the `IloExpr` class, given the current environment.
   (b) Use whatever code is needed to add the correct variables with the correct coefficients to this expression using the +, −, *, /, =, +=, −=, *=, and /= operators.
   (c) Add the constraint to the model using the `IloModel::add` method on a new `IloRange` object with the current loop's `IloExpr` object.
   (d) End the expression object to clear memory.

For example, suppose we have already defined an environment called `myenv`, a model called `mymodel`, and an array of variables of length n called `x`. Suppose we want to solve a standard form LP subject to equality constraints $Ax = b$, where A is an $m \times n$ matrix and b is a vector of length m, and each constraint takes the form $a_{i1}x_1 + a_{i2}x_2 + \ldots + a_{in}x_n = b_i$ for $i = 1, 2, \ldots, m$. Suppose that A, b, n, and m have all already been stored in memory as a two-dimensional array `A`, a one-dimensional array `b`, and integers `n` and `m` (so `A[i][j]` in our code is the $(i + 1), (j + 1)$-th element of A). Then we could use the following code to add all necessary constraints to the model:

```
for (int i=0; i<m; i++)
{
    // i = constraint number
    IloExpr myexpr(myenv); // empty expression
    for (int j=0; j<n; j++)
    {
        // j = variable number
        myexpr += A[i][j] * x[j]; // add each variable with the correct coefficient
    }
    mymodel.add(IloRange(myenv, b[i], myexpr, b[i])); // add assembled constraint
    myexpr.end(); // clear memory
}
```

For certain sparse problems (for example, network flows problems), it may be more efficient to simply specify all nonzero coefficients instead of constructing constraints row-by-row. In this case, we begin by defining an empty array of constraints from the `IloRangeArray` class. We can manipulate the coefficients of individual variables in each constraint using the `IloRange::setLinearCoef` method, and alter the bounds with the `IloRange::setBounds` method.

To define an empty array of constraints, use the following:

```
IloRangeArray mycon(env);
```

where `mycon` is its name and `env` is the environment. This defines an empty array whose elements are all `IloRange` objects. Individual constraints can be added using the `IloRangeArray::add` method as follows:

```
mycon.add(IloRange(env, lower, upper));
```

where `lower` and `upper` are the desired bounds. If several constraints all have the same set of bounds, we can quickly add several at a time using:

```
mycon.add(n, IloRange(env, lower, upper));
```

This will add `n` constraints with the same set of bounds. After completing this process, we will have an array of constraints with defined bounds but no variable coefficients. By not specifying an expression containing a linear combination of variables, their coefficients are assumed to be zero, so the next step is to individually set the nonzero coefficients. This is done using the `IloRange::setLinearCoef` method as follows:

```
mycon[i].setLinearCoef(var, coef);
```

where

- `mycon` is the name of the `IloRangeArray` object.
- `i` is the index of the constraint we wish to alter.
- `var` is the name of a variable.
- `coef` is the new coefficient to use for variable `var` in constraint number `i`.

Finally, we add the constraint array to the model using the `IloModel::add` method as usual.

Adding an Objective to the Model

Objectives are added to models in exactly the same way as constraints, using the `IloModel::add` method. The syntax is:

```
model.add(objective);
```

where `objective` is either an `IloMinimize` or an `IloMaximize` object, depending on whether the problem is minimization or maximization. The syntax for each is identical. To add a minimization objective to a model, we can use the syntax:

```
model.add(IloMinimize(env, expr));
```

where

- `model` is the name of the model to which the constraint is being added.
- `env` is the environment to which `model` belongs.
- `expr` is an expression that defines the cost function. Its syntax is the same as that of the `expr` argument of a constraint.

To make this a maximization, simply replace `IloMinimize` with `IloMaximize`. As with constraints, if the objective is short enough, it may be entered explicitly. Otherwise, it can be generated by defining an `IloExpr` object and constructing the objective term-by-term.

For example, suppose we have already defined an environment called `myenv`, a model called `mymodel`, and variables `x` and `y`, and that we want to define a minimization objective with cost function $x + 2y$. Then we could use the following:

```
mymodel.add(IloMinimize(myenv, x + 2*y));
```

Now suppose that instead of two variables we have `n`, stored in a variable array `x`, and that we want a cost function of the form $x_1 + 2x_2 + 3x_3 + \ldots + nx_n$. Then we could use the following:

```
IloExpr myexpr(myenv); // empty expression
for (int i=0; i<n; i++)
    myexpr += (i+1) * x[i];
mymodel.add(IloMinimize(myenv, myexpr));
myexpr.end(); // clear memory
```

As with the constraints, if the objective happens to be sparse, it may be easier to simply specify the nonzero coefficients. To do so, we begin by defining an empty objective object from the `IloObjective` class, and then use the `IloObjective::setLinearCoef` method to manipulate individual coefficients. Suppose we have already defined an array of 100 variables `x`, but the objective is simply to maximize $x_1 + 4x_{10} - x_{50}$. Then we could use the following:

```
IloObjective myobj = IloMaximize(myenv);
myobj.setLinearCoef(x[0], 1); // +x_1
myobj.setLinearCoef(x[9], 4); // +4x_10
myobj.setLinearCoef(x[49], -1); // -x_50
mymodel.add(myobj);
```

Extracting and Solving the Model

By this point, we should have an environment containing all desired variable definitions, and a model object containing all desired constraints and the objective. The next step is to extract this model to the CPLEX solver to obtain a solution, which can be done using an `IloCplex` object. To solve our model, we can use the `IloCplex::extract` and `IloCplex::solve` methods as follows:

```
IloCplex mycplex(env);
mycplex.extract(model);
mycplex.solve();
```

where

- `mycplex` is the name of our Cplex object.
- `model` is the name of the model to which the constraint is being added.
- `env` is the environment to which `model` belongs.

After calling the `solve` method, the extracted model will be sent to the CPLEX solver, which will attempt to find a solution. If running this program through the command line, the CPLEX solver's output will be shown while it looks for a solution. The method, itself, outputs a value of the `IloBool` data type with a value of `IloTrue` if the problem was found to be feasible, and `IloFalse` otherwise. This output can be used in our program to decide how to proceed.

Note that the `IloCplex` class contains a large number of methods for getting and setting the CPLEX solver options, but these will not be discussed here. These settings become particularly important when dealing with large-scale problems with integer constraints. For more information, see the documentation in the IBM Knowledge Center[5].

Exporting the Model Solution

After solving a model, information about the solution will be stored in the `IloCplex` object whose `solve` method was called. To actually access this information, we can use the `IloCplex::getStatus`, `IloCplex::getValue`, and `IloCplex::getObjValue` methods. To get the solution status, use:

```
cplex.getStatus()
```

This returns a string specifying one of the following:

- Optimal solution found.
- Feasible but not necessarily optimal solution found.
- Model known to be infeasible.
- Unknown whether any feasible solutions exist.

To get the value of an individual variable, use:

```
cplex.getValue(var)
```

where

- `cplex` is the name of our Cplex object.

[5] http://www.ibm.com/support/knowledgecenter/SSSA5P_12.6.2/ilog.odms.ide.help/refcppopl/html/classes/IloCplex.html

- `var` is the name of the variable in question. If the variables were part of an array, the array index should be included, as in `var[i]`.

To get the objective value, use:

```
cplex.getObjValue()
```

where, again, `cplex` is the name of the Cplex object.

Closing the Model

After associating a model object with a Cplex object, the Cplex object keeps track of changes that occur to the model, including addition or removal of constraints and changes to coefficients or ranges. In addition, after solving a model, the Cplex object maintains information about this solution. As a result, if we solve a model, then modify its constraints, then re-solve the model, CPLEX will use as much information as possible from the previous solution (for example, the basis) to quickly solve the modified problem. This is useful for many applications that require iteratively solving, modifying, and re-solving a model.

The `IloCplex::clear` method clears the current solution from a given Cplex object. This may be necessary if we are writing a program in which several different models will all be extracted to the same Cplex object. Its syntax is:

```
cplex.clear();
```

where `cplex` is the name of the Cplex object. This method should be called between each set of model extractions/solutions.

Finally, before exiting the program the environment should be cleared using the `IloEnv::end` method:

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

where `env` is the name of the environment object. This frees the memory allocated to all extractable objects within the environment.

3 Full Examples

This section contains several simple examples for putting the above concepts into practice, including all code needed for implementation. If in need of more complicated examples, the CPLEX Optimization Studio download includes a folder of example class files. For me, they are located in:

`C:\Program Files\IBM\ILOG\CPLEX_Studio127\cplex\examples\src\cpp`

A brief description of all example files can be found through the IBM Knowledge Center[6].

3.1 Hard-Coding Constraints

Consider the following simple example problem:

$$\begin{aligned} \max \quad & 5x + 15y \\ \text{s.t.} \quad & x + y \leq 200 \\ & x + 4y \leq 400 \\ & x, y \geq 0 \end{aligned}$$

The optimal solution is $x = \frac{400}{3} \approx 133.33$ and $y = \frac{200}{3} \approx 66.67$, with an objective objective value of $\frac{5000}{3} \approx 1666.67$. The following program defines a model to represent this LP, exports the model to CPLEX to obtain a solution, and then prints the results.

```
#include <iostream>
#include "ilcplex\cplex.h"
#include "ilcplex\ilocplex.h"
using namespace std;

int main()
{
    // Model Definition
    IloEnv myenv; // environment object
    IloModel mymodel(myenv); // model object
    IloNumVar x(myenv, 0, IloInfinity, ILOFLOAT); // variable x on [0,infinity)
    IloNumVar y(myenv, 0, IloInfinity, ILOFLOAT); // variable y on [0,infinity)
    mymodel.add(x + y <= 200); // constraint x+y <= 200
    mymodel.add(x + 4*y <= 400); // constraint 5x + 15y <= 400
    mymodel.add(IloMaximize(myenv, 5*x + 15*y)); // objective max 5x + 15y

    // Model Solution
    IloCplex mycplex(myenv);
    mycplex.extract(mymodel);
    IloBool feasible = mycplex.solve(); // solves model and stores whether or ...
        // not it is feasible in an IloBool variable called "feasible"
```

[6] http://www.ibm.com/support/knowledgecenter/SS9UKU_12.4.0/com.ibm.cplex.zos.help/Examples/topics/exampleCpp.html

```cpp
    // Printing the Solution
    if (feasible == IloTrue)
    {
        cout << "\nProblem feasible." << endl;
        cout << "x = " << mycplex.getValue(x) << endl; // value of x
        cout << "y = " << mycplex.getValue(y) << endl; // value of y
        cout << "cost = " << mycplex.getObjValue() << endl; // objective
    }
    else
        cout << "\nProblem infeasible." << endl;

    // Closing the Model
    mycplex.clear();
    myenv.end();

    // wait for user to press a key, otherwise the readout will immediately ...
        // disappear from the command line
    cout << "\nPress [Enter] to continue..." << endl;
    cin.get();
    return 0;
}
```

Running this program results in the following command line output:

```
Tried aggregator 1 time.
No LP presolve or aggregator reductions.
Presolve time = 0.01 sec. (0.00 ticks)

Iteration log . . .
Iteration:  1   Dual infeasibility  =   0.000000
Iteration:  2   Dual objective  =   1666.666667

Problem feasible.
x = 133.333
y = 66.6667
cost = 1666.67

Press [Enter] to continue...
```

3.2 Automatically Generating Constraints

Consider the following example problem:

$$\begin{array}{rl} \min & x_1 + x_2 + x_3 + x_4 \\ \text{s.t.} & 2x_1 + x_2 + x_3 + x_4 \geq 1 \\ & x_1 + 2x_2 + x_3 + x_4 \geq 1 \\ & x_1 + x_2 + 2x_3 + x_4 \geq 1 \\ & x_1 + x_2 + x_3 + 2x_4 \geq 1 \end{array}$$

The optimal solution is $x_1 = x_2 = x_3 = x_4 = \frac{1}{5} = 0.2$, with an objective objective value of $\frac{4}{5} = 0.8$. We could write a program similar to the previous example, individually defining each variable and constraint, but the constraints here are so similar that they can be easily generated using `for` loops. The following program uses variable and constraint arrays to solve the above LP.

```
#include <iostream>
#include "ilcplex\cplex.h"
#include "ilcplex\ilocplex.h"
using namespace std;

int main()
{
    const int n = 4; // number of variables/constraints

    // Model Definition
    IloEnv myenv; // environment object
    IloModel mymodel(myenv); // model object
    IloNumVarArray x(myenv, n, -IloInfinity, IloInfinity, ILOFLOAT); // array ...
        // of n unbounded variables x

    // Constraint Generation
    for (int i=0; i<n; i++)
    {
        IloExpr myexpr(myenv); // empty expression
        for (int j=0; j<n; j++)
        {
            if (j == i)
                myexpr += 2*x[j]; // coefficient 2 on diagonal
            else
                myexpr += x[j]; // coefficient 1 elsewhere
        }
        mymodel.add(IloRange(myenv, 1, myexpr, IloInfinity)); // i-th constraint
        myexpr.end(); // clear memory
    }
    // Objective Generation
    IloExpr myexpr(myenv); // empty expression
    for (int i=0; i<n; i++)
        myexpr += x[i]; // summing all variables
    mymodel.add(IloMinimize(myenv, myexpr)); // adding minimization objective
    myexpr.end(); // clear memory

    // Model Solution
    IloCplex mycplex(myenv);
    mycplex.extract(mymodel);
    mycplex.solve();

    // Printing the Solution
    for (int i=0; i<n; i++)
        cout << "x(" << i+1 << ") = " << mycplex.getValue(x[i]) << endl;
    cout << "cost = " << mycplex.getObjValue() << endl;

    // Closing the Model
    mycplex.clear();
    myenv.end();

    // wait for user to press a key, otherwise the readout will immediately ...
        // disappear from the command line
    cout << "\nPress [Enter] to continue..." << endl;
    cin.get();
    return 0;
}
```

Running this program results in the following command line output:

```
Tried aggregator 1 time.
No LP presolve or aggregator reductions.
Presolve time = 0.00 sec. (0.00 ticks)

Iteration log . . .
Iteration:  1   Dual objective  =   0.500000
x(1) = 0.2
x(2) = 0.2
x(3) = 0.2
x(4) = 0.2
cost = 0.8

Press [Enter] to continue...
```

By writing our code in this way, we can easily change the number of variables and constraints to solve a larger version of the above problem. The general version is

$$\begin{array}{ll} \min & x_1 + x_2 + \ldots + x_n \\ \text{s.t.} & \begin{bmatrix} 2 & 1 & \cdots & 1 \\ 1 & 2 & \cdots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \cdots & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \geq \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} \end{array}$$

for any $n \in \mathbb{N}$. The example above was the case of $n = 4$. To see the results for any other number, replace the value of `n` in the code. For example, for $n = 9$ we have:

```
Tried aggregator 1 time.
No LP presolve or aggregator reductions.
Presolve time = 0.01 sec. (0.01 ticks)

Iteration log . . .
Iteration:  1   Dual objective  =   0.500000
x(1) = 0.1
x(2) = 0.1
x(3) = 0.1
x(4) = 0.1
x(5) = 0.1
x(6) = 0.1
x(7) = 0.1
x(8) = 0.1
x(9) = 0.1
cost = 0.9

Press [Enter] to continue...
```