

# Object-oriented declarative model specification in C++

Risto Lahdelma\*

\*University of Turku

Department of Information Technology  
Joukahaisenkatu 3, FI-20520 TURKU, Finland  
Risto.lahdelma@cs.utu.fi

## Abstract

Mathematical optimization models are commonly implemented by specifying them in a high-level declarative modelling language. These languages allow specifying the overall model structure separately from the actual numerical values that are needed to form a specific instance of the model. The existing modelling languages are typically interpreted when the model is generated. In large-scale linear and mixed integer programming the interpretation and generation time may be substantial compared to the time to solve the model. This paper presents MPLC++, which is an object-oriented mathematical modelling language based on C++ classes and objects. Because the model classes are standard C++, they can be compiled and executed efficiently when the model is generated. This approach is particularly suitable in embedded time-critical on-line applications of mathematical optimization models.

**Keywords:** declarative programming, object-oriented mathematical modelling, linear programming, mixed integer programming, C++

## 1 Introduction

Mathematical optimization models are commonly implemented by specifying them in a high-level declarative modelling language. Examples of such languages or environments are GAMS /BKM88/, UIMP /EIMi82/, GXMP /Do186/, AMPL /FGK90/, MPL /Kri91/, PDM /Kri91a/, LPL /Hür93/ and MME /Lah94/. These languages allow specifying the overall model structure separately from the actual numerical values that are needed to form a specific instance of the model. Surveys of modelling languages can be found for example in /GrMu92/, /Kui93/ and /BCLS92/.

Most existing modelling languages require a separate interpreter environment or a pre-processor. Sometimes the modelling language is integrated with a solver, but often the solver is a separate program accepting some intermediate format. Thus, when models are embedded in various design, planning and control applications, the resulting system consists of several programs, which exchange data through large model files. Alternatively it may be possible to link the modelling language interpreter directly into the application. The problem with the above approaches is the overhead caused by

- inter-process communication through files,
- interpreting the modelling language,
- re-generating the internal model format for the solver,
- post-processing and transferring the results back to the application and
- re-initialising the solver in successive optimisation runs with only slightly different parameters.

In large-scale linear and mixed integer programming this overhead may be substantial compared to the time to actually solve the model.

This paper presents an implementation of an object-oriented mathematical modelling language MPLC++ in C++ using classes and instances. Because the model classes are standard C++, they can be compiled and executed efficiently to generate model instances. This approach is particularly suitable in embedded time-critical on-line applications of mathematical optimization models.

## 2 Object-oriented modelling language in C++

MPLC++ is a high-level, hierarchical, object-oriented modelling language, which can be efficiently implemented in C++ using objects and classes. MPLC++ supports modelling with generic, re-usable model components in a way similar to MME (**Mathematical Modelling Environment**) /Lah94/. Model parameters can be inline-coded in C++ or retrieved from auxiliary sources, e.g. external databases. Models can be generated directly into the internal storage format of an integrated optimizer, optimised and the results can be directly retrieved by the application. The basic elements of MPLC++ are implemented as C++ classes according to the class hierarchy shown in Table 1.

Table 1. The basic elements of MPLC++

Class	Abstract syntax
<b>Model</b>	Name Set(Var) Set(Param) Set(Model) Set(Constr)
<b>Constr</b>	Name Cexp <= Linexp <= Cexp
<b>Linexp</b>	Var   Cexp   Linexp (+   -) Linexp   Linexp (*) Cexp
<b>Var</b>	Name Min Max Unit
<b>Real</b>	
<b>Integer</b>	
<b>Binary</b>	
<b>Cexp</b>	Param   Val   Cexp (+   -   *   /) Cexp
<b>Param</b>	Name Val Unit

### 2.1 Models

An MPLC++ model consists of a name, a set of decision variables, a set of parameters, a set of constraints between decision variables and parameters and a set of submodels. A model is an implicit relation between decision variables and parameters. New models and submodels are derived from the predefined class **Model**. Decision variables, parameters and submodels may be specified in the private, protected and public section or in the constructor according to the needs for access control. Constraints are defined in the class constructor embedded in executable C++ code. Example:

```
class MyModel : public Model {
    Var x;          // private decision
                  // variables, parameters
                  // submodels

public:
    Var y, z;      // externally accessible
                  // variables, parameters
                  // and submodels

    // constructor
    MyModel(char name[], int n) {
        // local variables, parameters and
        // sub models
        ...
        // definition of constraints
        ...
    }
    // other methods
}
```

Models can be instantiated from model classes with different arguments. Models instantiated from inside another model become components of the surrounding model and a tree-structure of models is formed. The C++ dot notation can be conveniently used for specifying a path to the desired component. If submodels are instantiated from outside any model, they become independent models, forming thus a forest of models.

```
MyModel m1("m1", 2), m2("m2", 5); // two
independent models m1 and m2
```

Sometimes it is convenient to write a root-model directly into a C++ program without defining a class. The model scope is introduced using the **MODELSCOPE**-preprocessor macro, which accepts the name of the model as a parameter. The model scope is closed using the **ENDMODEL** macro.

```
MODELSCOPE(m); // Name=m
... // definition of decision variables and
    // parameters
... // instantiation of submodels,
    // definition of constraints

    MyModel m1("m1", 2), m2("m2", 5);
// submodels m.m1 and m.m2

ENDMODEL;

m.maximize(m.m1.z+m.m2.z); // optimise
linear objective
```

### 2.2 Decision variables

Associated with a decision variable are a symbolic name, a range of allowed values and the unit in which it is expressed. Decision variables are defined

directly in C++ programs as instances of the **Var** class or the derived classes **Real**, **Integer**, **Binary**. For example:

```
Var x("x", 0, 100, "kg"); // Name="x",
Min=0, Max=100, Unit="kg"
Real y("DB001"), z; // Min=0, Max=INF,
Unit= ""
```

Here *x* becomes a continuous decision variable in the range[0,100], measured in kilograms. The name *x* occurs twice in the previous definition. The C++ variable *x* is defined as a reference to a decision variable with run-time name "x". The C++ name *x* can be used in the succeeding C++ code where constraints and submodels are defined. The run-time name "x" is stored in a symbol table. Without a run-time symbol table it would be impossible to interpret a character string as a variable or to attach meaningful names to a listing of variable values. The C++ name and run-time name may differ as with *y* in the previous example. If the run-time name is omitted, the system will automatically generate a unique run-time name.

All arguments are optional. The unit defaults to an empty string signifying a dimensionless quantity. Real and Integer variables are by default non-negative, which is common in mathematical programming. Binary variables are by default 0/1 variables.

### 2.3 Parameters

Parameters are named symbolic constants, which can be defined directly in C++ programs as instances of the Param class or possible subclasses thereof. Example:

```
Param a("a",10,"g"); // Name="a", Val=10,
// Unit="g"
```

When parameters are used instead of ordinary C++ constants or variables, the system is able to keep track of how the model depends on the parameters. This makes it possible to support various parametric analysis techniques. It is also possible to implement spreadsheet-like incremental model re-generation techniques when some parameters are modified.

### 2.4 Constraints

We consider here only linear constraints. Constraints may have a C++ name and a run-time name. Linear constraints can be double inequalities with constant left and right hand sides or a single inequality or equality between two linear expressions. Unit compatibility is automatically checked in expressions and constraints. Many standard units are built into the system, but new units and conversion rules can also be defined. Incompatible units result in run-time errors. Unit conversions can be used for scaling expressions. For example

```
Constr b("b"); // name "b" for
0 <= 2*y-3*z <= 9; // double inequality
1+y >= 2-z; // unnamed constraint
x == 10*(y+z); // run-time error,
// unit mismatch
x == a*(y+z); // ok because
// a = 10g = 0.010kg
x/Unit("g") == 10*(y+z); // x in lhs is
// scaled by 1000
```

## 3. Extensions

The C++ inheritance mechanism can be used with MPLC++ in different ways. Model classes can be derived from more advanced model classes than the base class Model. Such model components will inherit decision variables, parameters, submodels and constraints from the parent model. Also more specialised variable and parameter classes can be easily derived from the predefined classes.

There is a need for built-in variable and parameter arrays, symbolic index sets and sequences in MPLC++.

## 4. Conclusions

It is possible to implement a readable, high level modelling language using C++ classes and pre-processor macros. The complete embedding of MPLC++ into C++ makes it easy to write integrated modelling and optimisation applications. Because all syntactic processing is done by the C++ compiler, and the resulting model can be directly generated in memory into the internal data structures of a solver, this approach should be very suitable for development of embedded applications with high performance requirements.

## References

- /BCLS92/ Bharadwaj A., Choobineh J., Lo A., Shetty B.: Model Management Systems: A Survey; Annals of Operations Research, vol. 38, December 1992
- /BKM88/ Brooke A., Kendrick D., Meeraus A.: GAMS A User's Guide; The Scientific Press, Redwood City, California 1988
- /Dol86/ Dolk D.: A Generalized Model Management System for Mathematical Programming; ACM Transactions on Mathematical Software; Vol 12 No 6; June 1986
- /EIMi82/ Ellison E.F.D., Mitra G.: UIMP: User Interface for Mathematical Programming; ACM Transactions on Mathematical Software; Vol. 8, No. 3, September 1982
- /FGK90/ Fourer R., Gay D.M., Kernighan B.W.: A Modeling Language for Mathematical Programming; Management Science, Vol. 36, No. 5, May 1990

- /GrMu92/ Greenberg H.J., Murphy F.H.: A Comparison of Mathematical Programming Modeling Systems; Annals of Operations Research, vol. 38, December 1992
- /Hür93/ Hürlimann T.: LPL: A mathematical programming language; OR Spektrum; Band 15, Heft 1, Springer-Verlag 1993
- /Kri91/ Kristjansson B.: MPL Modelling System User Manual, Maximal Software Inc., Iceland 1991
- /Kri91a/ Krishnan R.: PDM: A knowledge-based tool for model construction; Decision Support Systems; Vol. 7, No. 4, 1991
- /Kui93/ Kuip C.A.C: Algebraic Languages for Mathematical Programming; European Journal of Operational Research, Vol. 67, No. 1, May 1993
- /Lah94/ Lahdelma R.: An Object-Oriented Mathematical Modelling System; Acta Polytechnica Scandinavica, Mathematics and Computing in Engineering Series, No. 66, Helsinki 1994, 77 p.