FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master Thesis in Informatics

Adding C++ Support to MBEDDR

Zaur Molotnikov

FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master Thesis in Informatics

Adding C++ Support to MBEDDR

C++ Unterstützung für MBEDDR

Author:	Zaur Molotnikov
Supervisor:	Dr. Bernhard Schätz
Advisor:	Dr. Daniel Ratiu
Date:	September 15, 2013

Ich versichere, dass ich diese Diplomarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 15. September 2013 Zaur Molotnikov

Acknowledgments

If someone contributed to the thesis... might be good to thank them here.

Abstract

An abstracts abstracts the thesis!

Contents

Outline of the Thesis

Part I: Introduction and Theory

CHAPTER 1: INTRODUCTION

This chapter presents an overview of the thesis and it purpose. Furthermore, it will discuss the sense of life in a very general approach.

CHAPTER 2: THEORY

No thesis without theory.

Part II: The Real Work

CHAPTER 3: OVERVIEW

This chapter presents the requirements for the process.

Part I.

Introduction and Theory

1. Introduction

Here I introduce the main concepts and existing work, relevant to this Master Thesis, as well as present the motivation for the topic.

1.1. Projectional Editing

This section compares the traditional approach to build textual editors for the program code with the projectional approach, bringing up motivation for the least.

1.1.1. Traditional Approach

Traditionally programming languages are used in a textual form in text files, forming programs. However the textual nature is not typical for the structure of programs themselves, being rather low-level representation of code. Parsers are used to construct so called abstract syntax trees (ASTs) from the textual program representation. ASTs are structures in memory, usually graph-alike, reminding sometimes control flow graph, where nodes are different statements and edges are the ways control passes from one statement to the next one.

For the developer, using an editor, the degree to which the editor can support the development process is important. For this, the editor has to recognize the programming language construction and provide possible assistance. Among such assistance can be code formatting, syntax validation, source code transformations (including refactoring support), code analyses and verification, source code generation and others. Many of these operation rely indeed on the higher than text level notions related to program such as method, variable, statement. A good editor has to be aware of these higher level program structure.

Nowadays most of the editors work with text, and to provide assistance to programmer integrate with parser/compiler front-end for the programming language. This way to extract the program structure during editing is not perfect for several reasons. First of all, the program being edited as text is not syntactically correct at every moment, being incomplete, for example. Under this circumstances the parsing front-end can not be successfully invoked and returns error messages which are usually false-positive. Secondly, after minor editing of the code, usually the whole text file has to be processed again. Such compiler calls are usually computationally expensive, they slow down, sometimes significantly, the performance of the developer machine. Various techniques exist to speed it up, including partial and pre- compilation, but the problem is still relevant to large extent.

Moreover, the textual nature of the code complicates certain operations additionally. As an example, we can take a refactoring to rename a method. Every usage of the method, being renamed, has to be found and changed. To implement it correctly an editor must take into account various possible name collisions, as well as presume a compilable state of the program to even start the refactoring.

Not to mention the parsing problem itself. Parsing a program in a complex language like C++ is a difficult problem, it involves the need to resolve correctly scoping and typing, templates and related issues, work with pre-processor directives incorporated in the code. In this regard different compilers treat C++ in a different way, creating dialects, which may represent obstacles for the code to be purely cross-platform.

Listing 1.1: Closing several blocks

```
class MyClass {
  void doSomething() {
    while(true) {
      try {
        // ...
        }
      catch(Exception e) {
      }
    }
  }
};
```

The textual representation of program code, involves the need in formatting and preserving syntax. These both tasks, indeed, have nothing to do with program functionality, and addtionally load the developer, reducing productivity. As an example, here I can mention the need to close several blocks ending at the same point correctly, indenting the closing brace symmetrically to opening one, and putting in the exact amount of braces. The Listing 1.1 demonstrates it in the last few lines.

1.2. Projectional Approach

Another approach which can be taken in the editor for a language is called projectional approach. Projectional editors do not work with a low-level textual representation of a program, but rather with a higher level concept, ASTs.

Working with ASTs directly has several advantages over the conventional text editing. Firstly, all syntax errors are no longer possible, as there is no syntax. Secondly, there is no need to format the code anyhow, since it is only typical for textual code. Thirdly, all features, which in textual approach require parsing, can be implemented without parser, because AST is always known to the editor.

Projectional editors have to display somehow the AST to the developer, in order for him/her to work with it. Such visualization of an AST is called "projection", giving a name to the editor class.

The model of code is stored as AST in the projectional editor. As in the Model-View-Controller pattern ([7]) the view for the model can be implemented separately. Thus the code may look in a many different ways to the user. For example, the AST can be visualized as a graph, similar to control flow graph. This visualization, however, is not always advantageous being sometimes not compact and complicated to overview.

```
MyClass::MyClass(int8 x):  from MyClass {
  if (x > 0) {
    while ( true ) {
      --x;
      if (x == 0) {
        return;
      } if
    } while
  } if
}
```

Figure 1.1.: Example projection of an AST, “source code” view

One of the well-accepted way to visualize AST is buy visualizing its textual representation, as if it would was written as a code in the programming language, see Figure 1.1. The statements in the projectional are only selected as whole. There is now way to just select the “while” word for cut or copy, without selecting the condition and the block belonging to the statement. This behavior represents the position of the condition and while-body in the AST as children of the while statement. The statement can be selected all, only with children.

Additionally, each block delimiters are just a part of block visualization. They are organized in a proper way automatically, and there is no way to delete or confuse them, as well as type them initially. Each closing brace is marked with the parent statement name (not a problem to implement such behavior in viewing AST), enhancing navigation through the AST's projection.

As one can see the projection to text in a projectional editor looks almost the same, as the conventional textual editor. This can cause some confusion for the developer at first, as attempts to edit this textual visualization as a real text will fail. Eventually, however, advantages of such visualization overwhelm the disadvantages. Among the benefits of the textual projection over text are quicker code construction after short learning, better way to select code fragments, since not individual characters or lines, but rather AST nodes or groups of nodes are selected, plus all the advantages, the projectional editing brings by itself, as discussed above.

1.3. C++ in Projection

The goal of this Master Thesis is to research different aspects connected with implementation of the C++ programming language in a projectional editor. As a tool base for this Jetbrains MPS has been selected together with the based on it mbeddr project. This section overviews the technologies.

1.3.1. Jetbrains MPS

Jetbrains MPS stands for Jetbrains Meta Programming System. In this IDE-like software it is possible to develop domain specific languages. The approach taken in the Jetbrains MPS is rather unique. One defines a language in it not through the canonical grammar approach, but instead through defining so called concepts, and relationships between them similar to those in ER diagrams. A logical part of a language can be made a concept. Related to C++, class, method declaration, field declaration can be represented as a concept.

Each concept can have children, which should be assigned with a role and cardinality. For example, the class concept can have children of method concept, with a role called "methods", and cardinality 0..n.

Concepts can form hierarchies as classes normally do in object oriented programming languages. For this each concept should have a base concept (inheritance), and can implement several interface concepts (interface implementation). Concepts can be abstract, for the use purely only in inheritance to create other non-abstract concepts with a common parent.

Each concept is described by several views on it. Among such views there are editor, behavior, constraints, type system, generator and few more views.

The editor view is designed to give a look for a concept, and a way to input it. This is where the projection of an AST node is defined. As editors are mostly defined to look like text, a program in the C++ programming languageimplemented in MPS looks almost like a regular C++ code.

The behavior view, can be used to define certain behavioral methods for a concept. A concept is represented there similar to a java class, and it is possible to define the methods in a Java-like language.

The constraints view can be used, to limit in a desirable way relationships the concept can have to other concepts.

The type system view is used mainly to define a type for each instance of a concept. The type is used later (for example, in the constraints), to determine suitability of the concept instance for a place, where it is used.

The generator view, as well as the textgen view, is used to define the way, the concept instance is going to be transformed when generation of the AST is invoked by the MPS user. In general, the generation is needed in the end of the programming, to obtain the source code from the AST in a textual representation. This textual representation is needed because all existing tools (like compilers) expect text code nowadays. It is worth mentioning that, in theory, one could implement a compiler, which works directly from the AST in the editor, eliminating thus the need in generators to text.

The representation of a new language created in concepts and views to them, present a seamless approach to creating a new language with a projectional editor. Thus Jetbrains MPS is a good suitable technology for the practical implementation of the Master Thesis goal.

1.3.2. mbeddr Project

The mbeddr project represent mainly an implementation of the C programming language in the Jetbrains MPS environment. Having embedded systems and software for them as a main focus, mbeddr provides certain language extensions to empower the programmer in the mentioned domain.

Being a different language the C++ programming language shares a lot of commonality with C. As Jetbrains MPS allows to some extent incremental language construction, as the basis for the C++ programming language implementation in Jetbrains MPS the mbeddr project was considered to be suitable.

The use of mbeddr however, could not be purely incremental, and required some changes to the mbeddr itself. The changes were introduced however, in a way to make mbeddr simply more extensible, without creating a dedicated branch working as a basis for the C++ programming language only.

1.3.3. Motivation

The selection of the topic for the Master Thesiscan be motivated in several ways.

At first, the author is not aware of any implementations of C++ editing in a projectional editor. The goal to research such opportunities for one of the most complicated, traditionally highly text-oriented, object oriented languages is new.

Secondly, extending the mbeddr project with the C++ programming language has a practical interest for the mbeddr project users. Being comparatively lightweight, the C++ programming language supports object-oriented programming paradigms and more. It is thus of a high interest for embedded domain developers, using mbeddr, to be able to have the C++ programming language in the arsenal.

Being an extension for the mbeddr project the discussed in this Master Thesis C++ implementation is following the general mbeddr goals. Among this goals are targeting the embedded programming domain, and making the language, when possible, more supportive to the user. Both goals have some influence on the implementation, which are discussed in the following chapters.

Part II.

C++ and Projectional Editing

2. C and C++

Initially C++ appeared to be an extesnion to the C language, called "C with Classes" [8]. Till now the high degree of commonality can be found between the two languages. Mainly, the most of the C code is going to be valid C++ code.

2.1. Reference Type and Boolean Type

Among primitive types and operations there are two major differences relevant to practice. They are discussed in the following subsections.

2.1.1. Reference Type

Reference type is basically a new construct in C++ which represent the old notion of pointer in C with a different syntax. The syntax for a variable of type `T` can be used with the variable of type reference to `T`, or `&T` as it is designated in C++. It comes especially handy when passing arguments to methods by reference, which avoids creating a copy of an object to pass by value.

Listing 2.1: Reference Type in a Copy Constructor

```
class MyClass {
    MyClass( const MyClass& original);
};
```

The value of the reference type in C++ is bigger however, than just a new syntax for pointers. A constant reference to a class object has to be used in constructor to give it a special meaning of a copy constructor, as the Listing 2.1 demonstrates.

2.1.2. Boolean Type

The C++ programming language has a special `bool` type to represent the two logical values. There is no such type in C.

No matter, the `bool` type is not present in C, for the convenience of the user the mbeddr C implementation introduced a type `bool`, which is translated to `int8_t`, together with `true` and `false` values, converting to `1` and `0` respectively. This implementation can be considered better than the original C++ `bool` type present in the generated code, because the C++ programming language standard does not define explicitly the size of the `bool` type [5].

In the context of embedded systems, it is often very important to know precisely, with which type the user is operating, as the limited resources are important to consider. Substituting the `bool` type with the `int8_t` type ensures that the size of the `bool` variables is known. Also, it can be changed as needed in the `TypeSizeConfiguration` in each model created with mbeddr separately.

The C++ standard explicitly allows the `bool` type to participate in integer promotions. This ensures further the compatibility of the custom-written text code, which may use actual `bool` type, with the code generated with substitution of `bool` to `int8_t`, as the user `bool` will be promoted.

Among limitations of this approach one can take as an example `std::vectoor<bool>`. Since the word "bool" itself is never generated, it is not possible to use the specialization of template, which ensures storage optimization, through, though, higher processing load when extracting a value from such vector.

2.2. Modules

1. Differences between C and C++
   a) Reference Type and Boolean Type
   b) Modules Support
   c) Object Oriented Approach Support

3. C++ Object-Oriented Programming

The C++ programming language is a multi-paradigm programming language. The ability to support the object-oriented programming, is incorporated via classes.

A class represents a new type in the C++ programming language. Each class may have data in the form of fields, and behavior in the form of methods. Two types of methods are special for C++ - constructors and destructors, they have special meaning and syntax.

Encapsulation is enabled via governing access permissions to fields and methods of a class. The access control is governed with the creation of `public`, `protected` and `private` class sections.

Inheritance is implemented in C++ via allowing for each class to have one, or even many base classes. Inheritance from a base class is performed under a certain access control modifier. There is no pure notion of an interface, but rather abstract classes are introduced.

Polymorphism is implemented via pointer-to-class type compatibility over inheritance-connected classes.

The implementation of these C++ features in a projectional editor environment is discussed in the following chapters.

3.1. Class declaration

The C++ programming language is a mature language, with long traditions, and high flexibility, [6] can serve as an example. Although the language is also famous for being complex. It will not be possible to simplify language, removing features from it, which will restrict the language use. In this work I try to research, how the editor can be more supportive for the user, to eliminate usual mistakes made while programming, as well as provide help in structuring the code.

3.1.1. Visibility Sections

Instead of declaring visibility type for individual class members, visibility sections are created in C++.

The sections can be opened with a string `private:`,`protected:` or `public:` within a class declaration, and closed when another section is opened or when the class declaration ends. This allows the user to open and close the same section multiple times and declare sections without any particular order.

Various coding guidelines ([1], [2]) exist to enforce some restrictions on the visibility sections.

In particular, the sections are allowed to be opened only once. This ensures, the reader of the code will see interface of the class (public section) in one place, “contents” of the class (private section) in one place, and opportunities to access members in the inheriting class (protected section) in one place, without the need to scroll through through the whole class declaration.

Another typical requirement in coding standards, is the order of the section. Usually the public section is required to be first, for the class users to see immediately the public interface, the class provides.

```
class WebPage {
  public:
    explicit WebPage() (constructor)
    WebPage(const WebPage& original) (copy constructor)
    boolean loadFromFile(string path)
    string getHtml()
  private:
    int32 visits
  protected:
    string html
}
```

Figure 3.1.: Sample class type declaration

The Figure 3.1, shows an example class declaration implemented in the projectional editor. The class concept has the visibility sections as children. Each section is given a separate role, and can appear 0 or 1 times. The editor for the class concept orders the visibility sections so, that the public section always comes first, followed by the private and protected sections.

The creation of a section is made with the use of so called *intention* in MPS. The user uses *Alt+Enter* combination on the class declaration to create visibility sections. It should be more practical and fast for the user, compared to typing the keyword, colon and indenting the result.

A question arises on how to support another way to represent a class, so that it will reflect requirements from a different coding standard. And as a way to resolve it a definition of another editor for a class concept can be offered, Unfortunately, the current version of Jetbrains MPS 2.5 does not support a definition of multiple editors for the same concepts. This limitation however is addressed in the newer 3.0 version.

3.1.2. Constructors

Constructors are special methods of a class, used to construct the class instances.

Constructors have special syntax and no return type, being similar to class methods. Additional value, however, the constructors gain, when participating in type transformations. Namely, when a constructor of a class B exist from a type T, instances of the type T can be used whenever the B class instance is required. The constructor will be *implicitly* called and a temporary object of class B is going to be created as a mediator.

Thus constructors extend the type system. Since this type extension can not be easily observed, it is highly possible to get various *run-time* errors or unexpected behavior.

Listing 3.1: Example of implicit constructor error

```
#include<iostream>
using namespace std;

/*
 * API definition
 */

class Circle{
private:
  int r;
public:
  Circle(int radius){
    r = radius;
  }

  float getPerimeter(){
    return 2*3.14*r;
  }
};

void print(Circle c) {
  cout << "The perimeter is: " << c.getPerimeter();
}

/*
 * Use case by user of the API
 */

int main(){
  print(5);   //Prints "The perimeter is: 31.4"
  return 0;
}
```

The Listing 3.1 demonstrates a simplified use case where the function `print()` is invoked on `int` without any compiler error, and the resulting behavior is unexpected.

To avoid similar, it is possible to deprecate participation of a constructor in type conversions, adding a word `explicit` to the constructor declaration.

The described problematic motivated the following decision. When a new constructor is created, it is by default declared to be explicit, the user must intentionally change it to get the type conversion behavior. Such behavior is safer by default.

3.1.3. Copying

In the C++ programming language the programmer controls memory allocation fully on his/her own. This affect the way of copying the object. In C++ a programmer should define two methods for a class: a copy constructor and an assignment operator. These two methods work when assignment like `a = b` happens, when instances of a class are passed by value to a function, when one object is initialized with a value of another object and so on.

C++ serves here sometimes dangerously generating default copy constructor and assignment operator, which by default represent a bitwise cloning of an object. This can lead to problems.

Listing 3.2: Need in custom copy constructor

```
class Resource{
  int* r;
public:
  Resource(){
    r = new int(0);
  }
  ~Resource(){
    delete r;
  }
};

int main(){
  Resource a;
  Resource b = a; // b.r is the same address as a.r
  return 0;
}
```

The Listing 3.2 demonstrates a program which crashes upon execution as destructors of `a` and `b` are deallocating memory with the same address, after default copy-constructor copies the address from `a` into `b`.

To avoid the described problem, the programmer has to either define proper copy-constructor or forbid copying of the objects of the class. The same applies to the assignment operator. Many standards require the two functions to be implemented in sync, [3]. This can be performed be implementing the assignment operator first and reusing it in the copy-constructor. To simplify the first, specialized macros exist, for example `DISALLOW_COPY_AND_ASSIGN` or `Q_DISABLE_COPY`, [2], [4].

The use of macros in C++ appears often in similar cases, in order to perform some language-engineering tasks to add the missing features to the language. Macros bear pure textual nature, and are processed by the pre-processor. Some negative effects may come out: need to pre-process reduces the speed of compilation and hides the resulting code from a programmer, macros lead to error prone programming, as no type checks are possible, macros make code less analyzable by automatic analyzers.

```
class A /copyable and assignable/ {
  public:
    explicit A() (constructor)
    A& operator = (const A& originial ) (makes class assignable)
    A(const A& originial) (copy constructor)
    int16 getX()
  private:
    int16 x
}
```

Figure 3.2.: Hinting about copyable and assignable class properties

The projectional editor allows for another solution, different from macros. In order to provide some support for the programmer regarding the copying issue, the projectional C++ implementation hints on the class declaration its assignable and copyable properties, Figure 3.2, and generates by default the declarations of copy-constructor and assignment operator.

The copy constructor and the assignment operator are detected automatically. Two intentions are provided on the Class concept to forbid or allow copying. The forbidding intention imitate the macros mentioned above, but displaying and explaining the implementation to the userFigure 3.3. The implementation consists of moving the declarations of two functions to the private section of the class. Implementation is not required for such functions.

```
class B /neither copyable nor assignable/ {
  public:
    explicit B() (constructor)
    int16 getX()
  private:
    int16 x
    B& operator = (const B& o ) (makes class not assignable)
    B(const B& o) (makes class not copyable)
}
```

Figure 3.3.: Class made not copyable by the *Forbid copying* intention

3.2. Encapsulation and Inheritance

3.3. Polymorphism

Part III.

Conclusion

Conclision goes here

Appendix

A. Detailed Descriptions

Here come the details that are not supposed to be in the regular text.

Bibliography

[1] Class layout recomendations, possibility.com. http://www.possibility.com/cpp/cppcodingstandard.h

[2] Google style recomendations for c++, http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml.

[3] Open office c++ coding standard, http://wiki.openoffice.org/wiki/cpp_coding_standards.

[4] Qt project documentation page, contains q_disable_copy macro. http://qt-project.org/doc/qt-5.0/qtcore/qobject.html.

[5] Standard for programming language c++ (c++11).

[6] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Professional, 2001.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[8] Bjarne Stroustrup. Classes: An abstract data type facility for the c language. *Sigplan Notices*, 17:41–52, 1982.