Task documentation CLS: C++ classes in PHP56 for IPP 2015/2016
Name and surname: Svätopluk Hanzel
Login: xhanze10

1 Assignment

The assignment of this project was to write a PHP56 script, which analyzes simple C++ classes' definitions and is able to generate reports on inheritance hierarchy or class members.

2 Solution

My solution is divided into several cooperating parts, each of them in a separate file and class. These classes have assigned namespaces, which allows us to use simple autoloading technique, eliminating need to use the `require` statements.
Now, let's look at some of these classes:

2.1 CPPAnalyzer

CPPAnalyzer is the main class of the project. It is also the first object to be created and has several tasks including parsing command line arguments and processing the data gathered during other states of the program.

2.1.1 Parsing command line arguments

This part was fairly easy. PHP5 allows us to use the `getopt` function to parse arguments, so there is no need to use regular expressions. My project however also allows the use of short version of the arguments from the assignment, so I had to do a simple normalization to get the right value.

2.1.2 Processing data and generating output

Second task of the CPPAnalyzer class was to process data gathered in other parts of the project and generate XML (using `XMLWriter`) corresponding to the input parameters.
The process itself is fairly simple. For the class inheritance diagram, classes are processed recursively. In the details mode, all members of all classes (or just one based on the argument) are processed in cycles.

2.2 Scanner

The main task of the *Scanner* class is to process the input file (either the one passed in as a command line parameter or `stdin`), various tokens, create their object (*GeneralToken*, *IdentifierToken*, *PrivacyToken*, *TypeToken*, *TypeSpecifierToken*).
The scanner itself is implemented as a simple finite state machine with just two states to recognize special characters, and words. Words are then divided into keywords and identifiers.

2.2.1 GeneralToken

Used for storing information about any simple token: comma, semicolon, parentheses, asterisks, ambersands, etc.

2.2.2 IdentifierToken

Used for all names, i.e. classes, members, function arguments are all identified using *IdentifierToken*.

2.2.3 PrivacyToken

PrivacyToken represents a keyword specifying privacy levels: *private*, *protected*, *public*.

2.2.4 TypeToken

This class represents all simple variable types: bool, char, char16_t, char32_t, double, float, int, void, wchar_t

2.2.5 TypeSpecifierToken

Objects of this type are all the keywords, that can be used before a *TypeToken*: unsigned, signed, long, short. This approach allows easier parsing.

2.3 Parser

The next state after scanning the file to tokens is to connect those tokens into higher objects. In my case these objects are classes with their respective members.

2.3.1 AbstractMemberSymbol

Abstract class to save information, which are duplicate for both attributes and methods: their type, name and whether they are static or not.

2.3.2 VariableSymbol

Objects of this class are used in two different situations: for class attributes and as arguments of methods. In retrospect I understand, this isn't the ideal solution as it doesn't follow the OOP methodologies, but because I didn't have more time to refactor this approach, I decided to let it be.
The class itself extends *AbstractMemberSymbol* and the only information it adds is the way to compute md5 hash, which is later used in class inheritance conflicts checking.

2.3.3 MethodSymbol

Another class extending the *AbstractMemberSymbol* class. This one however also adds some informations: its arguments (with both types and names), and whether the method is virtual and in case it is, whether it's purely virtual.

2.3.4 ClassSymbol

The key class in this project. It wraps all the information about it's members, their privacy levels, inheritance, etc. It also provides the methods necessary for inheriting, deciding the abstraction of the class and more.
The abstraction itself is solved in three simple steps:

1. Accumulate all properties and methods from all classes
2. Check for conflicts using md5 hashes of these members
3. If no conflicts were found, extend the class by adding these members to a special array in the class

   During this step privacy levels may be shifted according to the inheritance type.