A programming tool to ease modular programming with C++

J. Baltasar García Perez-Schofield
Dpt. of Computer Science, University of Vigo
Edif. Fundición, s/n. Campus As Lagoas-Marcosende
Vigo, España
jbgarcia@uvigo.es

Francisco Ortín
Dpt. of Computer Science, University of Oviedo
Calvo Sotelo s/n. 33007.
Oviedo, España
ortin@uniovi.es

Abstract— **Module management support is very rough in the C and C++ programming languages. Modules must be separated in interface and implementation files, which will store declarations and definitions, respectively. Ultimately, only text substitution tools are available, by means of the C/C++ preprocessor, which is able to insert an interface file in a given point of a translation unit. This way of managing modules does not take into account aspects like duplicated inclusions, or proper separation of declarations and definitions, just to name a few. While the seasoned programmer will find this characteristic of the language annoying and error-prone, students will find it not less than challenging. In this document, a tool specially designed for improving the support of modules in C++ is presented. Its main advantage is that it makes it easier to manage large, module-based projects, while still allowing to use classic translation units. This tool is designed for students who have to learn modular programming; not only those in the computer science discipline, but also those in other engineerings in which programming is part of the curriculum.**

Keywords: Learning, module management, preprocessor, compiler, C++.

I. INTRODUCTION

The C++ programming language has its origin in C, which itself has its origin between 1969 and 1973 [16]. From that date of birth, the C programming language has undergone various changes, firstly by its author, Dennis Ritchie, later by an ISO standardization committee (ISO JTC1/SC22/WG14). Similarly, C++ was born at the beginning of 1990 decade, created by Bjarne Stroustrup [17], and taking the existing C programming language at that time as basis. It has evolved heavily, firstly due to its author, and lately by an ISO standardization committee (ISO JTC1/SC22/WG21) [19].

Since other, more recent programming languages have not had such a long history (Java [15], or C# [9]), they have successfully solved many problems found in C and C++, avoiding them due to a fresh design start (a fresh start has the advantage of peventing backwards compatability problems). In fact, the author of C++ has himself made various proposals regarding the simplification of his programming language [20], empowering new characteristics such as the use of STL library or the explicit support of modular programming itself and making programming simpler or homogeneous, meaning that ambiguities are kept to a minimum.

A. Motivation

As in many other computer science faculties (for instance, [4]), we have selected C++ to be the language of choice for the first courses. While in the first course it is preferred over C because of some higher abstractions (for instance, argument passing by reference), it does also share the same basic syntax with many other programming languages. Although C-like syntax is often criticized, its use is broadly considered an advantage, as learning the syntax part of the programming language is time saved for other future programming languages, such as Java and C#.

C++ was actually not designed for education: it was designed for providing to C a higher programming paradigm such as object-oriented programming, while maintaining its performance. C syntax was also designed to be minimal, not simple or easy to understand. Thus, it is not surprising that its characteristics have a large room for improvement from an educational point of view, while still being a highly successful programming language.

One of the main challenges we have detected students must face with is module management, in which C++ provides no better abstraction than C. While in other programming languages, such as C#, or Java, modules are specified with high detail, and even automatically managed by programming environments [5], in C or C++ modules are simply source code scattered in various files (called translation units in C terminology). The rules regarding how programs should be factorized in translation units are clearly specified; however, the relationship among the modules of the application and the translation units which compose that application are not stated [4].

An example illustrating the common problems when facing modular programming in C++ follows. The objective of such a simple example would be to create a very simple module in which the *PI* constant and the *sqr()* function are defined. The programmer will have to firstly understand that, in C and C++, there is a separation of modules between the so called header file (declarations, a *.h* file), and the implementation file (definitions, a *.cpp* file). It is a good practice to divide declaration and definition parts of modules, clearly stating the interface of each module with the rest. However, the mandatory division in two files is not even common currently. This is just done in C++ that way because of the primitive mechanisms available at the time when it was designed. This

This work has been partially funded by the Department of Science and Technology (Spain) under the National Program for Research, Development and Innovation; project TIN2008-00276 entitled “Improving Performance and Robustness of Dynamic Languages to develop Efficient, Scalable and Reliable Software”.

mechanisms for module programming support are just simple text substitution. After all, many other modern programming languages support modular programming. They emphasize module interfaces, without the necessity of creating two different files (their linkers are able to extract that information automatically).

Next, students will have to understand that the implementation file needs to include its own interface as well, for just in case the constant *PI* is used inside it.

```
// math.h
#ifndef MATH_H
#define MATH_H
const double PI = 3.1415927;
double sqr(double x);
#endif
```

```
// math.cpp
#include "math.h"
double sqr(double x)
        { return x * x; }
```

Finally, students will have to create and use the so called header guards (MATH_H in the code above), for the case in which the header is included more than once in a single translation unit (that would duplicate the existence of any class declared or constant defined in the header file, the very case of PI above). There are workarounds for avoiding this problem in this very simple example, (though it involves learning the meaning of the *extern* modifier), but header guards are required more often than not, so the beginner will have to understand the (very low-level) basics of the preprocessor, and get used to creating them for any header file.

In this document, the *Cp^3*-- tool is presented, allowing to program in a simplified C/C++ subset, specifically in regard to module management. The remaining parts of the document are presented after this section: firstly, the state of the art is discussed, and then the main characteristics of the preprocessor are detailed. Some cases of study are then explained so it is possible to study the main advantages of the tool. Finally, results and initial performance measurements are shown, following the conclusions.

II. The help given by the module manager prototype

Although this module manager was designed for the standard ISO C++ programming language, the authors took two main design decisions, for the sake of simplicity. The first one was to simplify data declarations, avoiding unneeded and undesirable ambiguity (inherited from C). Along with this decision, it was also stated that the C++ subset chosen for this dialect should be left unchanged, i.e. apart from the obvious differences due to the utility of this tool itself, there shouldn't be any difference in the programming language. The learning curve would thus be as smooth as possible. The objective is for novel programmers to find it more homogeneous and more coherent than standard C++, which would again be translated in a smooth learning curve. Seasoned C++ programmers should be able to get full advantage of the system in a matter of minutes, while novel ones could avoid C++ complexities.

The complexities of the programming language have been exposed repeatedly in time. Frequently, it has been said that a subset of C++ is trying to get out from a language [20] that is so firmly rooted in C, thus exporting a very complex syntax. For instance, it is possible to use signed and unsigned characters; it is possible to mark as constant a pointer, the content it is pointing to, or even both... these and similar other characteristics are interesting in some specialized contexts, but from the educational point of view, they are confusing and definitely not simple nor homogenous (from an educational point of view, characters should just be able to represent all characters the program can support; and also, marking a value as constant should have one and only one meaning).

The main difference between standard C++ and this dialect is the mandatory use of *namespaces*. While these are optional in C++, design issues suggest the benefits of using them (avoiding name clashing by means of making it possible to arrange code in independent scopes, [18][19], introducing the need of a more comprehensive support for modules, in order to complement them). A modular version of C++, aimed at students must obligatorily empower its higher-level characteristics. When low-level C++ features are really needed, a modular approach is probably not the best one. In that cases, programmers will be facing with device driver programming or similar projects; that is why the tool was designed to share its existence with standard C++ programming, instead of trying to substitute it all.

More emphasis has been put in two major uses of the programming language for module creation: utility (function-based) modules and classes (classed-based modules). First ones are basically the best use of procedural programming, regardless of whether object-oriented programming is used or not. Second ones are the expected in object-oriented programming in C++, except for the fact that it is not possible to create objects along with the class declaration.

A more detailed discussion on this topic is given elsewhere [7]. Besides, this tool can be found in the authors' website[1].

A. *Basic cases of study*

C++ is a hybrid programming language, supporting both procedural and object-oriented programming. The support of procedural programming is done through the wrap of a version of the C programming language, meaning that many programs written in C can be compiled without modifications in C++. The main difference between the C programming language and the C version of the programming language included in C++ is that the latter provides better compile-time verification mechanisms, mainly by adding type-safety to the original C. This means that C++ is more suitable, even for procedural programming only, for students than plain C [4][20], as C++ provides them with better error checking, and true (or, at least, at a higher level of abstraction) parameter passing by reference for functions (this is actually very important, since it allows to avoid pointers in introductory courses).

1) *Procedural programming*

Probably the simplest example of module is the one that is composed by functions, such as the algorithm standard module [11]. In C++ there are two ways for encapsulating simple functions: the object-oriented way of creating them as static function members inside a class, and the procedural one (since 1998) of putting them inside a *namespace.*

1 *http://webs.uvigo.es/jbgarcia/prjs/cp3/*

A really simple example could be the *maths* module, briefly presented in the introduction, in which the *PI* constant and some functions (such as *sqr()*) are going to be provided.

There are some variations around this example. The first one would consist of abusing the separation of declarations and definitions, putting the definition of *PI* in the header only. The preprocessor guard constant (MATH_H) is explicitly used to avoid duplicated definitions, in the case of multiple inclusions, so it would compile correctly in any case. The programmer could also declare some functions as *inline.* This module is so simple and the difficulties about its construction are so few, that the weak module support mechanisms of C++ are clearly shown as one of the main obstacles novice programmers must face to.

The *Cp*^{3}-- version, quite simpler, is shown below. A module is a single *.mpp* file. Students do only need to concentrate in creating the module, marking the public interfaces and hiding the implementation details, without having to worry about the module management restrictions of C++. It is worth noting that the programming language is nearly left untouched: the *inline* keyword marks whether this function should be marked as an inline function, or in the case of member functions, shoule be stored in the header file (with the same resulting consideration). Also, the constant *PI* is declared in the header file with an extern automatically, while the actual definition will lie in the implementation file.

```
// math.mpp
namespace Math {
        const double PI = 3.1415927;
        inline double sqr(double x)
                { return x * x; }
        inline
        double addPercentage(double val, double x)
                { return ( val * ( 1 + x ) ); }
        inline
        double removePercentage(double val, double x)
                { return ( val - ( val * x ) ); }
}
```

The presented source code is a single *Cp*^{3}-- file that will be translated to the previous files, automatically.

2) *Object-oriented programming*

The Person module uses the previous module Math shown above, and presents two simple attributes, and various methods returning attributes in the *Person* instance. Follows the source code for ISO 1998 standard C++.

```
// person.h
#ifndef PERSON_H
#define PERSON_H
#include <string>
#include "math.h"
namespace Person {
class Person {
public:
        Person(const std::string &n, double s)
                : name( n ), salary( s ) {}
        double getSalary() const
                { return salary; }
        double getNetSalary() const
                { return Math ::
                 removePercentage(salary, 0.20); }
        const std::string &getName() const
                { return name; }
        virtual std::string toString() const;
private:
        std::string name;
        double salary;
};
}
#endif
```

The previous source code fragment is the interface of this translation unit (.h file), which was logically designed as a module. Though again some variations are possible (such as declaring *toString()* as inline, and defining them in the interface file), this is probably the most classical translation of this module.

```
// person.cpp
#include "person.h"
#include <iostream>
#include <sstream>
#include <cstdlib>
std::string Person::Person::toString() const
{
        std::ostringstream out;
        out << getName() << ", " << getSalary();
        return out.str();
}
```

```
// main.cpp
#include "person.h"
#include <iostream>
int main()
{
        Person::Person p( "Baltasar", 1800 );
        std::cout << p.toString() << std::endl;
        return EXIT_SUCESS;
}
```

The implementation file just gives a body to the *toString()* member function. However, the complexity and number of lines of code devoted to workaround the lack of explicit support for modules in C++, as well as the knowledge required to know how to divide the code, is also the primary concern here. Although the seasoned C++ programmer has gotten used to this schema, that doesn't mean it could not be done using a higher abstraction.

There is also a lack of support for changes, profiling and maintanability: in case the programmer decided to make setter methods *inline,* it would involve porting valuable amounts of code from one file to the other one. It must be taken into account that in ISO C++ a method defined in the declaration of the class is automatically inline, even when this is not explicitly marked, which can led to confusion. This is solved in *Cp*^{3}-- in which there is not any difference between defining a member inside the declaration. Thus, those members which are *inline* must be always explicitly declared as *inline*. While these decisions might be trivial for the seasoned programmer, beginners will probably follow the approach of trial and error.

By means of the *Cp*^{3}-- tool, it is possible to translate the following source code of the *Person* module to the source already presented above (the *main.cpp* file would be left untouched). Note that the generated code is not thought to be human-readable, nor also to be the main source code for the project. The only, human-readable, source code for any project would be the set of *.mpp* files (any change to the *cpp* or *h* files will be lost the next time the module is compiled).

```
// person.mpp
#include <string>
#include <iostream>
#include <sstream>
#include <cstdlib>
#include "math.h"
namespace Person {
class Person {
public:
        Person(const std::string &n, double s)
                : name( n ), salary( s )
                {}
        inline double getSalary() const
                { return salary; }
        inline double getNetSalary() const
                { return Math::removePercentage
                                ( salary, 0.20 ); }
        inline const std::string &getName() const
                { return name; }
        virtual std::string toString() const
                {
                  std::ostringstream out;
                  out << getName()
                      << ", " << getSalary();
                  return out.str();
                }
private:
        std::string name;
        double salary;
}
}
```

It is possible to put the *getters* member functions and the *toString()* member function, as shown, to be *inline*: it would just be a matter of preceding them with the *inline* keyword. In fact, it is possible to essay the best approaches, by trial and error, effortlessly. The presented source code is a single *Cp*^{3}-- file that will be translated to both previous files, automatically. Again, it must be remarked that the involved process is highly tedious and error-prone, as it is a very low-level mechanism.

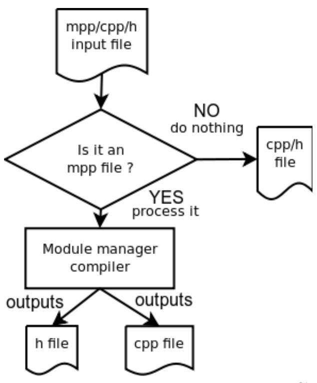

Figure 1: Processing scheme for *Cp*^{3}--.

B. Implementation

This tool makes it possible to combine C++ translation units with modules in the same project, as shown in Figure 1. The tool simply ignores files that are not a module.

Cp^3-- is therefore employed as a preprocessor, but it is technically implemented as a compiler/translator. The input file is parsed completely, apart from methods implementation, which is left to the actual C++ compiler. The tool inserts appropriate #line compiling directives so it is assured that possible error messages will be correctly tracked to the actual input file, instead of the generated files.

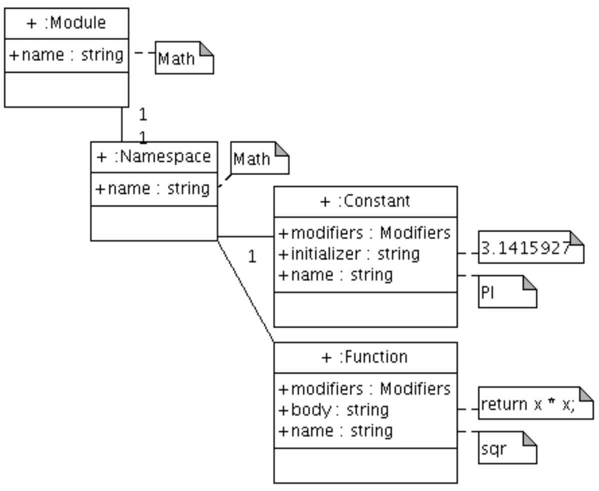

Figure 2: An example abstract tree built by *Cp*^{3}--.

Figure 2 represents the abstract syntax tree for the *maths* example shown before. *Cp*^{3}-- does only read the input file once, building all data structures alongside the parsing. Once the tree has been built, it has been ensured that basic error checking has been performed as well. In the next phase, the tree is checked out following the *Visitor* pattern, and thus performing additional checks, matching the strictness level chosen by command line switches. If no error is found, the last phase is triggered, in which the entities in the tree are visited again in order to generate the source code for both interface and implementation files [2].

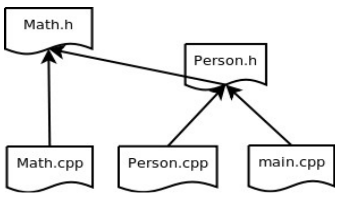

Figure 3: Dependencies among files in ISO C++.

Thus, error detection can be done before generating code, and is indeed carried out automatically by the tool, even allowing for the different levels of strictness discussed above. For example, all methods must have an implementation (unless they are pure virtual); modifiers, such as *const* or *static*, must be correctly typed; their combinations must also be legal. This way, a constant variable member is forced to be declared static, as well as it must be static in order to be allowed to be initialized. In procedural mode, only constants and functions are allowed, and therefore a global variable will be marked as erroneous.

The purpose of the tool presented here is centered at simplifying module management for C++. Even for the simple example presented here, Figure 3 presents a set of fairly complex relationships among interface and implementation files, something that a seasoned C++ programmer is aware of, but overwhelming for a student. Figure 4 shows the equivalent relationships using *Cp^3*--.

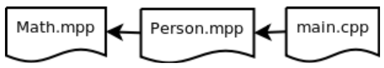

Figure 4: Dependencies among files using the tool.

III. State of the Art

There are many other developments in research of learning, such as IOPL [8], which is able to cover C++, or BlueJ [12] [13][14], covering Java. However, none of them cover the specific issue of making modular programming simpler.

There are also new programming languages, such as D [1] [3][6], which try to ease C++ programming, including modular programming. Unfortunately, these new programming languages still have to prove themselves and gain popularity to be broadly considered for education.

The C++ standardization committee has produced at least five documents about supporting modules in C++, of which we would like to refer to revisions 2 and 5 [21]. In the first one, a mechanism more or less similar to the one studied here is drafted. However, the standardization committee proposal evolved to a more ambitious mechanism in revision 5. One of the clearly stood objectives are to decrease compilation times. In this state, the standardization committee has chosen to approach from a totally different point of view (more similar to *Modula-2*). This contrasts to the simpler, namespace-centric approach taken here, in which one of the preconditions of the work was to keep syntax changes to a minimum. In that sense, the work of the committee seems to have gotten apart from our interests. Even worse, there is still no agreement about modular programming in the C++ standard committee, and apparently will not make it in the next standards release, codenamed ISO C++ 200x.

Preprocess [10], is an unpublished tool that more or less uses the same approach taken here. However, the author is not concerned about macros, does not support namespaces and certainly his objective is not to obtain a clean, simple standard-compliant schema. In general, it does not even completely cover the ISO C++ 1998 [11][19].

LZZ [22] is another tool like Preprocess. However, it is at the other side of the spectrum, trying to completely compile the language. It therefore needs a careful installation process. For example, in order to generate the #line directives, just for the case there is a compiling error, special command line arguments must be written. The includes directory for C and C++ must also be provided so it is able to check the existence of the headers included. On the other hand, the use of namespaces is not mandatory, so identifiers in a module pollute the main namespace. Finally, it also includes extensions for the C++ programming language, something completely out of the scope of this project.

IV. Results

A seminar was organised for undergraduate students, in an advanced subject, for all of them wishing to test this tool. The subject is called “Object Technology”, in the Computer Science degree at the University of Vigo. This Test was performed on course 2008-2009.

Before and after the seminar, a pretest and a posttest was delivered to students in order to check, basically, whether they a) had found the system helpful for the understanding of modular programming, b) had found the system useful for learning, c) which characteristics would they improve. More than twenty students answered these tests for year 2009. A copy of the tests, as well as a comparison between the results obtained in pretest and posttest is found elsewhere [7].

The results of the questionnaires are quite encouraging, as a wide majority of the answers suppose a high degree of satisfaction. These tests were done to a set of undergraduate students, which had a seminar of two hours in which they were taught on using the system for twenty minutes, and finally had to complete some exercises.

Some of the questions were repeated in both tests, in order to study the change in opinion after working with the system. Their opinion about their own knowledge was important, so their were asked about how deep they thought their knowledge was about modular programming. In the pretest, more of the 90% answered they had some or advanced knowledge. This percentage decreases in around a 10% in the posttest, giving interesting details: there is an increasement in the number of students saying they have some knowledge, while the number of students with deep knowledge decreases in about a 38%, This can be explained because of the weak training in modular programming students receive, specially when they study C++. Other languages, such as Java, make the use of modularity mandatory, but many times this is transparently managed by an integrated environment, and somehow this probably makes them unaware of code factorization taking place, or at least, having any benefit.

About the question of whether they thought this programming technology was useful for teaching, students answered with more than a 90% that it was useful, with no significant changes.

A control question was also put in both tests in order to check whether they have understood the concepts around the tool. More than a 80% answered the correct question in both tests, and more than a 90% answered this one and another possible correct option (out of five possible answers). The percentage of students answering “I don't know”, decreased from 10% to 0% in the posttest.

Another question was related to the usefulness of modular programming. Again, students answering “I don't know” decreased from 5% to 0% in the *posttest*, while the thought of modular programming being useful in theory and practice increased from around a 75% to an 85%, as well as there was an increase of students thinking of modular programming being useful for teaching.

The remaining questions were put there in order to know what they thought about the prototype. The first one was related to whether they thought the use of a prototype would be an important tool for their improvement in their studies. More of an 80% answered positively to this question, which is a good result, given that more answering options were given in the *posttest.* The only significative percentage of opinion of these other options were the answer “I don't know”, with, however, less than 10 points.

More than a 60% thought that the use of the tool was simple, while the remaining students though its complexity was average (not simple nor complex). No one thought the prototype was difficult to use.

About whether their perception of modular programming had changed after the use of the prototype, more than a 71% recognized it was changed to some extent or even a lot.

The last two questions were presented in order to get their opinion about modular support in C++. The first one shows how they think (>85%) that C++ should have a better modular programming support, while the last one shows with more than a 90% that it would be better to avoid including new syntax or constructions.

V. Conclusions

Module management support for C++ is very rough. Though seasoned programmers are already used to this matter, students find this need of separation of interface and implementation in two parts (as well as the involved procedures, such a macro header guards) strange and unnecessary complex.

Actually this process, once mastered, becomes a mechanical (though tedious) habit, thus making it possible to automatize it. In this document, a prototype of a tool for accomplishing that task, the *Cp*^{3}-- module manager, has been presented.

In order to try to prove that modular programming could be more appealing for beginners, a seminar was given for undergraduate students. The feedback obtained from the questionnaires used then, was used in order to improve the tool.

The modifications made to the language are really minimal: the same keywords have the same meaning in the same context. The only difference is that now, everything is unified in a single translation unit which will be automatically translated in the interface and implementation files required. Beginners will understand the programming language better, while seasoned programmers can adapt without trouble in a matter of minutes. It is also worth noting that traditional code can be mixed in the very same project.

A tool based in the idea brought by this prototype could be easily and transparently added to the C++ compiler tool chain. It does not suppose any overhead nor quality decrease for the generated code, as its use would only have a slight impact in compile time. This would be the ideal situation, since such a decision would mean that the benefits of a tool such as this one would be available without the need of installing more software or worrying about using standard C++ r not. However, it is still possible to employ it as a prototyping preprocessor, aiming at providing the first approach to the source code that is actually needed. In that case, a code formatter will be needed, obtaining that way working code without syntactic errors.

References

[1] Alexandescu, A. The D Programming language. Addison-Wesley Professional; 1 edition (to be published in May, 2010). ISBN 978-0321635365.

[2] Appel, A. W. Modern Compiler Implementation in Java: Basic Techniques. Cambridge University Press (January 13, 1997). ISBN 978-0521586542

[3] Bell, K., Ivar Igesund, L., Kelly, S. Parker, M. Learn to Tango with D. Apress 2008. ISBN 1590599608

[4] Biddle, R. L., Tempero, E. D. “Teaching C++. Experience at Victoria University of Wellington”. IEEE Transactions on education 1995. ISSN 0 8186 5870 3/95.

[5] Böck, H. The Definitive Guide to the NetBeans Platform 6.5. Apress 2009. ISBN: 978-1-4302-2417-4

[6] Digital Mars. The D Programming language. http://www.digitalmars.com/d/. Access time: 2005 to 2009.

[7] García Perez-Schofield, Baltasar. “Cp3--, a C++ programming language preprocessor for Module Management”. Technical Report IT1/2009 in the Department of Computer Science, University of Vigo, 2009.

[8] Harrison, C.J., Sallabi, O.M., Eldridge, S.E.. An initial object-oriented programming language (IOPL) and its implementation. IEEE Transactions in education, VOL. 48, NO. 1, February 2005

[9] Heljsberg A., Wiltamuth S., Golde, P.. The C# Programming Language. Addison-Wesley Professional (October 30, 2003). ISBN 0321154916.

[10] Hohmuth, M. http://os.inf.tu-dresden.de/~hohmuth/prj/preprocess/. Access time: January 2010

[11] Josuttis, M. The C++ Standard Library: A Tutorial and Reference. Addison-Wesley Professional; 1 edition (August 22, 1999). ISBN 0201379260.

[12] Kölling, M.; Rosenberg J. “Blue—A language for teaching object-oriented programming,” in Proc. 27th Special Interest Group on Computer Science Education (SIGCSE) Tech. Symp. Computer Science Education, 1996, pp. 190–194.

[13] Kölling, M.; Rosenberg, J. “An object-oriented program development language and a software development environment suitable for environment for the first programming course,” in Proc. 27th SIGCSE Tech. Symp. Computer Science Education, 1996, pp. 83–87.

[14] Kölling, M.; Quig, B.; Patterson, A.; Rosenberg, J. The BlueJ System and its Pedagogy. Computer Science Education, 1744-5175, Volume 13, Issue 4, 2003, Pages 249 – 268

[15] Naughton, P.; Schildt, H. Java 2: The Complete Reference. Osborne/Mcgraw Hill Media Group 1999; 3rd edition. ISBN-10: 0072119764

[16] Ritchie, D.. The Development of the C Language. Proceedings of the Second History of Programming Languages conference, Cambridge, Mass., April, 1993. Later collected in the History of Programming Languages-II ed. Thomas J. Bergin, Jr. and Richard G. Gibson, Jr. ACM Press (New York) and Addison-Wesley (Reading, Mass), 1996; ISBN 0-201-89502-1

[17] Stroustrup, B. The C++ Programming Language. Addison-Wesley, 1986.

[18] Stroustrup, B. The design and evolution of C++. Addison-Wesley, 1994.

[19] Stroustrup, B. An overview of the standard C++ programming language. Handbook of Object technology. CRC press LLC Boca Ratón, 1998.

[20] Stroustrup, B. Learning Standard C++ as a New Language. Technical report in AT&T Labs, 2000.

[21] Vandevoorde, D. Modules in C++ (revisions 2 and 5). Std. committee documents N1778=05-0138, 2005 and N2316=06-0176, 2007.

[22] LZZ. http://www.lazycplusplus.com/, accessed in July 2010.