Informe técnico / *Technical Report* **IT1/2009**

Cp³--, un preprocesador del lenguaje de programación C++ para gestión de módulos

Cp³--, a C++ programming language preprocessor for Module Management

Palabras clave: Programación orientada a objetos, lenguajes de programación, programación modular.
***Keywords**: C++, Object-oriented programming, programming languages, modular programming*

Resumen: El soporte para programación modular es muy pobre en el lenguaje de programación C++. En realidad, sólo hay disponibles herramientas de sustitución de texto, a través del preprocesador de C++, para cualquier proyecto. En este documento, se presenta una herramienta especialmente diseñada para el soporte de módulos. La principal ventaja de esta herramienta es que permite la gestión de proyectos de gran envergadura, con módulos, a la vez que permite la programación tradicional en C++, lo cuál es útil para usar en código antiguo, o mezclar código antiguo con código nuevo. Esto funciona, símplemente, porque el preprocesador sólo trata aquellos archivos con extensión .cp3/.mpp. Además, permite seleccionar el nivel de rigidez en cuanto al uso de técnicas orientadas a objetos.

***Abstract:** Module management support is very poor in the C++ programming language. In the end, only text substitution tools are available, through the C++ preprocessor, in order to support modules in a given project. In this document, a tool specially designed for supporting modules in C++ is presented. The main advantage of this tool is that it allows to manage large, module-based projects, as well as it allows programmers to still use C++ in the same way they are used to, for example for legacy code, as this tool just acts as a translator that creates the appropriate header and implementation files,with a .cp3/.mpp extension. Finally, it allows to enforce object-oriented programming by preventing some programming techniques to be used (though this can be adjusted through command-line options).*

Área de Lenguajes y Sistemas Informáticos
Departamento de Informática
Universidad de Vigo

Cp3--
C++ Preprocessor for Module Management

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

Abstract. Module management support is very poor in the C++ programming language. In the end, only text substitution tools are available, through the C++ preprocessor, in order to support modules in a given project. In this document, a tool specially designed for supporting modules in C++ is presented. The main advantage of this tool is that it allows to manage advanced projects, as well as it allows programmers to still use C++ in the same way they are used to, for example for legacy code, as this tool just acts as a translator that creates the appropriate header and implementation files, provided the extension of the file is .cp3/.mpp. Finally, it allows to enforce object-oriented programming by preventing some programming techniques to be used (though this can be adjusted through command-line options).

1 Introduction

The module management capabilities of the C++ programming language are nearly zero (Stroustrup, 1986). The main basis is the linker, allowing to connect a function call through different translation units.

The remaining mechanisms are mere text substitution tools, residing in the preprocessor. Follows a discussion of these possibilities: a) file inclusion. This is carried out by the preprocessor directive `#include`. This directive involves the substitution of the whole file for the line in which it is found; b) preprocessor conditionals, allowing to detect whether the header file has been already compiled or not (`#ifndef`); c) programmers knowledge, which makes programmers to divide their code in interface files (headers) and implementation files (cpp files). However, the C or C++ programming languages never suggest or give an unique way to do this.

It is worth noting that in ISO C++ 1998 there are two kind of files: header files, with the 'h' extension, and implementation files, with the 'cpp' extension. The former one is expected to contain function prototypes and constant declarations, while all the implementations should be put in the latter one (this changes a little bit when using the *inline* implementation for methods).

Finally, the way to divide code in this two kind of files is fairly mechanical when one learns how to do so, and therefore introduces the possibility of being automated by the compiler itself, or at least by an auxiliary tool such as the one introduced here.

The remainder of this paper is organized as follows: in the next section, a simple case is shown, so the value of the proposal can be fully understood. Then, the complete set of transformations, as well as the limitations of the C++ dialect supported are discussed. Finally, conclusions and future work are presented.

2 A case of study

The objective of this tool is to allow better (in terms of simplicity and a high-level abstraction) modular programming support in C++. Firstly, programmers do not need to learn a new language nor new constructions, as the C++ language has been left untouched, though only an (intended) subset of it is supported. Finally, this approach can let programmers merge modules created with this tool with regular, already existing standard header or implementation C++ files, so transition

becomes as smooth as possible.

The following examples present simple cases of modules that are going to be easily generated by Cp^3.

2.1 Utility modules

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

A really simple example could be the *maths* module, in which we are going to provide the PI constant and the *sqr()* function. Firstly the standard C++ version is presented.

```
// math.h
#ifndef MATH_H
#define MATH_H

namespace Math {
	const double PI = 3.1415927;

	double sqr(double x);

}

#endif
```

```
// math.cpp

#include "maths.h"

double Math::sqr(double x)
{
	return x * x;
}
```

The variations around this example consist of declaring the constant to be *extern*, which would allow the programmer to drop the use of the preprocessor constants. The programmer could also mark the function 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 programmers must face to.

The Cp^3 version, quite simpler, is shown below. Programmers must just concentrate only in creating the module, without having to worry about C++ limitations. It is worth noting that the programming language is left untouched: the *inline* keyword marks whether this function should be marked as an *inline function*, or in the case of member functions, in the header file (with the same resulting consideration). Also, the constant 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;
	}
}
```

The presented source code is a Cp[3] file that will be translated to the previous files, automatically. As stated before, the involved process is so mechanic that it is very simple to automatize it.

2.2 Object-oriented programming

This simple module, *Person*, presents a constant, *DefaultName*, and various getters and setters. Follows the source code for standard C++.

```
// person.h
#ifndef PERSON_H
#define PERSON_H
#include <string>

namespace BussinessLogic {

class Person {
public:
	static const std::string DefaultName;
	static const std::string DefaultSurname;

	Person(const std::string &s = DefaultSurname,
		const std::string &n = DefaultName)
		: name( n ), surname( s )
		{}

	const std::string &getName() const
		{ return name; }

	const std::string &getSurname() const
		{ return surname; }

	const std::string &getEmail() const
		{ return email; }

	unsigned int getAge() const
		{ return age; }

	const std::string &getAddress() const
		{ return address; }

	std::string toString() const;

	void setAge(unsigned int a);
	void setAddress(const std::string &a);
	void setEmail(const std::string &e);
private:
	std::string name;
	std::string surname;
	std::string address;
	unsigned int age;
	std::string email;
```

```
};

}

#endif
```

The previous source code fragment is the interface of this module. Though some variations are possible (such as declaring setters as *inline*, and defining them in the interface file), this is probably the most classical translation of this module.

```
#include "person.h"

const std::string BussinessLogic::Person::DefaultName    = "John";
const std::string BussinessLogic::Person::DefaultSurname = "Doe";

void BussinessLogic::Person::setAge(unsigned int a)
{
      age = a;
}

void BussinessLogic::Person::setAddress(const std::string &a)
{
      address = a;
}

void BussinessLogic::Person::setEmail(const std::string &e)
{
      email = e;
}

std::string BussinessLogic::Person::toString() const
{
      std::string toret = getSurname();

      toret += ", ";
      toret += getName();
      toret += " (";
      toret += getEmail();
      toret += ")";

      return toret;
}
```

The implementation of this module just gives a body to setters, as well as the *toString()* member function. However, the weight of the code related to workaround the lack of 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 gets used to this schema, that doesn't mean it could not be done easier. There is also a lack of support for changes: if the programmer decided that setters could be *inline*, it would involve porting valuable amounts of code from one file to the other one.

By means of the Cp^3 tool, it is possible to translate the following source code for the module to the source presented above. 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 coude for any project would be the .mpp files (any change to the cpp or h files will be lost the next time the module is compiled).

```
// person.mpp
#include <string>
namespace BussinessLogic {
class Person {
public:
      static const std::string DefaultName;
      static const std::string DefaultSurname;

      inline Person(const std::string &s = DefaultSurname,
                    const std::string &n = DefaultName)
            : name( n ), surname( s )
            {}

      inline const std::string &getName() const
            { return name; }

      inline const std::string &getSurname() const
            { return surname; }

      inline const std::string &getEmail() const
            { return email; }

      inline unsigned int getAge() const
            { return age; }

      inline const std::string &getAddress() const
            { return address; }

      std::string toString() const {
            std::string toret = getSurname();

            toret += ", ";
            toret += getName();
            toret += " (";
            toret += getEmail();
            toret += ")";

            return toret;
      }

      void setAge(unsigned int a) {
            age = a;
      }

      void setAddress(const std::string &a) {
            address = a;
      }

      void setEmail(const std::string &e) {
            email = e;
      }
private:
      std::string name;
      std::string surname;
      std::string address;
      unsigned int age;
      std::string email;
}
}
```

We can even decide to put the setters member functions, as shown, to be *inline*: it would just be a matter of preceding them with the *inline* keyword. The presented source code is a cp^3 file that will be translated to the previous files, automatically. Again, the involved process is so simple (although tedious and error-prone) that it should be carried out by the compiler or an auxiliary tool (the very case presented here).

3 The C++ dialect supported

While this module manager was designed for the standard C++ programming language, the author decided to actually drop some of the possibilities of the language mainly those related to the extreme flexibility of C for variable declaration, as they include unneeded and undesirable ambiguity. 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 should be for seasoned C++ programmers to be able to get full advantage of the system within of minutes.

The main difference between standard C++ and this dialect (modular C++, *mC++* from now on), is the mandatory use of *namespaces*. While these are optional in C++, design issues suggest the benefits of using them (Stroustrup, 1998).

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 (Stroustrup, 2000) that is so firmly rooted in C, resulting in very complex syntax, such as the existence of different syntax variations for variable declarations, to put an example.

A modular version of C++ must obligatorily limit the spureous possibilities of the language: possibly, when they are really needed, a modular approach is probably not the best one. In that cases, programmers will be facing with device driver programming and other low level application projects.

3.1 Preprocessor macros

Preprocessor macros are simply not allowed. From the preprocessor, only the *#include* directive is left, with the same meaning it normally has, as the intention was to avoid changing the language at all. This is actually the reason preventing changing this for some meaningful keyword, such as *using* (probably like in *using module*), already present in the programming language.

A more high-level of abstraction directive, *import*, already present in the language, is available, with no actual changes to the *#include*, as it is translated to that preprocessor tool. It can also coexist with the standard form. The motivation for this was only to remove any hint about the *cpp*, the C++ preprocessor being used at all.

```
// math.mpp

namespace Math {
	const double PI = 3.1415927;

	inline double sqr(double x) {
		return x * x;
	}
}

// person.mpp
#include <iostream>
```

```
#include <string>
import Math;

using namespace std;

namespace BussinessLogic {
class Person {
public:
	static const std::string CanonicalName = "John Doe";
	// more things...
}
};
```

3.2 Namespaces

Namespaces are supported exactly as they are presented in standard C++. The only difference is the addition of support for the *public* and *private* labels.

```
namespace A {
	static void bar() {
	}
	void foo() {
	}
}
```

In the case shown above, the use of "static" means private, while its absence means "public". This behaviour is supported, though a more high-level, intuitive fashion is added:

```
namespace A {
public:
	void foo() {
	}
private:
	void bar() {
	}
}
```

When an inner namespace is found in the private section of a given namespace, then its members are private, not public. Also note that neither of these mechanisms can be applied to classes, only to objects.

A side note should be taken into account about the *using* directive. The reasons for this can be shown in this example:

```
#include <string>
using std::string;

namespace StrUtil {
	std::string &rtrim(std::string &x)
	{
		/* more here... */
	}
```

```
	std::string &ltrim(std::string &x)
	{
		/* more here... */
	}

	std::string &trim(std::string &x)
	{
		string &aux = ltrim( x );
		return rtrim( aux );
	}
}
```

While the *using* directive declares that the class *string* can be used without being fully qualified, it is still needed to use *std::string* in the function member declaration. This is because it was decided to avoid the inclusion of any *using* directive in headers. The inclusion of *using* in headers would avoid any possibility of identifier isolation. As soon as a using directive was used, it would mean the mandatory use of that identifier in case of a missing full qualification, and even worse, the possibility of identification clash. That would make the use of *namespaces* totally ineffective.

3.3 Functions

Functions are allowed due to the hybrid (i.e., not pure) nature of C++. It is not uncommon to find functions outside classes even in modern C++ (for example, in the *algorithm* module), as many times the use of a simple function eliminates unnecessary hassle. This is covered in other languages with the use of classes that are just wrappers for various static member functions (a possibility still available in standard C++ but unneeded due to the presence of *namespaces*).

However, when a function is created as a member of *namespace*, then that *namespace* can only contain other functions, not classes. The reverse case is also true: one class can be declared in one *namespace*, and nothing else. The strictness in applying these rules can be adjusted through command-line options, however, as discussed later.

The only modifiers allowed here are *static* (meaning private visibility within the *namespace*), and *inline*, as a petition to substitute the code of the function instead of creating a function call. These are the same meanings they have in the very same context in C++.

3.4 Constants

Constants are allowed as members of a *namespace*. The modifier *const* is mandatory here. Constants can be found in any module, no matter whether there is a class or a set of functions defined there.

3.5 Classes

A big effort has been put into parsing classes correctly. All classes can be written, in general, as if there weren't actually two files to refer to, acting, therefore, transparently. Insights are given in the following sections.

```
namespace foo {

class A {
private:
```

```
      int x;
public:
      void foo() {
            cout << "Hello" << endl;
      }
};

}
```

The semicolon at the end of the class is optional. In any case, there is no support for creating objects directly in that part of the class declaration. Object creation is achieved from inside methods in the usual way, or as members of the namespace:

```
namespace foo {

class A {
private:
      int x;
public:
      void foo() {
            cout << "Hello" << endl;
      }
};

A obja;

}
```

3.5.1 Member functions

Member functions (or methods), are actually the main reason of this whole work (probably along with static member fields, and plain functions). The very same modifiers are used in methods, being the only difference the inclusion of the code in the same file, along with the member declaration.

There are, however, some issues that are worth noting.

```
class Counter {
private:
      int count;
public:
      Counter(int inic = 0)
            { count = inic; }
      inline int getCount() const
            { return count++; }
};
```

The *inline* keyword is now mandatory if the programmer wants to have the code of a method substituted instead of being called when used. Note that this is still a hint for the compiler: the use of the inline keyword, or the inclusion of a method in the class declaration does not guarantee that the function will be *inlined*. The final decision is always taken by the compiler.

3.5.2 Constructors and destructors

Constructors are full supported, including the explicit keyword, and the quick initialization list. Also, as any other method in the class, the *inline* keyword is supported, meaning that the constructor will be included inside the class declaration, and therefore *inlined* when used. Destructors are full supported, without any change.

```
class Counter {
private:
      int count;
public:
      inline explicit Counter(int inic = 0) : count(inic)
            {}
      virtual ~Counter()
            {}
      inline int getCount() const
            { return count++; }
};
```

3.5.3 Member fields

Member fields do not vary at all. However, the static ones can be initialized inside the class, homogeneously. Indeed, it would be an error to initialize any member field: the semantics of constructors are left unchanged.

```
class Person {
public:
      static const unsigned int MaxAge = 120;

      Person(const std::string &n, int a)
            : name( n ), age( a )
            { if ( age > MaxAge ) {
                  throw std::runtime_error( "impossible age" );
            } }
      ~Person()
            {}
      inline unsigned int getAge() const
            { return age; }
      inline const std::string &getName() const
            { return name; }
private:
      std::string name;
      unsigned int age;
};
```

3.5.4 Inheritance

Inheritance is absolutely untouched. A class can inherit from any other class, with the appropriate visibility modifiers (*public*, *protected* and *private*).

```
class Employee: public Person {
public:
      Employee(const std::string &n, int a, double w)
            : Person( n, a ), wage( w )
            {}
      inline double getWage() const
            { return wage; }
private:
      double wage;
};
```

3.5.5 Encapsulation

There isn't any change to encapsulation within classes. Examples about encapsulation and visibility have been shown along with this section.

3.5.6 Polymorphism (late binding)

In standard C++, polymorphism is (syntactically) achieved through the use of the keyword '*virtual*', and sometimes the colophon "= 0". This is because in standard C++ a method can be written directly inside the class (implying the *inlining* of the function member), or outside it, (living normally in a separate, implementation file). The mentioned colophon syntactically differentiates the pure virtual function (without implementation) from the latter, when they don't have a body.

For the sake of simplicity, however, *mC++* does not allow nor needs the colophon anymore, as all methods have their implementations besides them, in a single file. If a member function is declared *virtual* and does not have a body, then it is understood as a pure virtual function.

The standard C++ example would therefore be:

```
// Figure.cpp
class Figure {
public:
      virtual double calculateArea() = 0;
};
```

While the same example using *mC++* would be:

```
// Geometry.Figure.mpp
namespace Geometry {
      class Figure {
      public:
            virtual double calculateArea(); // "= 0" is optional
      }
}
```

4 Behaviour of Cp³

The default behaviour of Cp³ discussed above, can be modified to be more or less flexible. Depending on the command line option passed to the preprocessor the following changes will be applied.

Option	Effect
No command line option	Same as `-level=3`.
`--level=3`	All limitations.
`--level=2`, `--level=1`	Functions and classes can be mixed in the same namespace. More than one class allowed per namespace, more than one namespace allowed per main (outer) namespace.
`--help`	Copyright message and usage.
`--force`	Avoid time stamp checking

5 Related work

The standardization committee has produced at least five documents about supporting modules in C++, of which I would like to refer to revision 2 (Vandevoorde, 2005) and revision 5 (Vandevoorde, 2007). In the first one, a mechanism more or less similar to the one studied here is discussed. However, the standardization committee is more ambitious about supporting modules than the current approach. One of the clearly stood objectives are to decrease compilation times. This made the proposal evolute to the current state (revision 5), the second one mentioned above. In this state, the standardization committee has chosen to approach from a totally different point of view (more similar to Modula-2), out of the namespace-centric approach taken here.

Preprocess (Hohmuthm 2004), 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.

6 Satisfaction 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. Test were performed on course 2008-2009.

Before and after the seminar, a *pretest* and a *posttest* (a corresponding version of both tests translated into English is included in appendixes B and C) 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 comparison between the results obtained in *pretest* and *posttest* is found in Appendix A.

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, 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 this 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 though this programming technology was useful for teaching, students answered with more than a 90% that it was useful, with no significative changes.

A control question was also put in both tests in order to check whether thay 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. 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 though of modular programming being useful in theory and practice increased from around a 75% to an 85%, as well

as there was an increasement 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 thougth its average complexity was average (not simple nor complex). No one thought the prototype was difficult to use.

An interesting question was what students liked and disliked about the prototype. More than an 85% liked that it was simple to use, and that it automated the use of modules in C++. About the characteristics they disliked, it was the very same answer with a 33% of students (probably they thought it was too simple), while nearly a 62% disliked the restriction of use of some characteristics of C++, which motivated the change in favour of improved flexibility commented in past sections.

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 an 90% that it would be better to avoid including new syntax or constructions.

7 Performance

Initial measurements have been performed in order to have an proximate idea of the overload that the preprocessor supposes for complete compilation. The test consisted in compiling a program generated by the module manager one hundred times, and then do it again but executing the preprocessor before. The obtained results are summarized in table 1. The first row corresponds with a single module (*Person2*) being processed, while the second row stands for a module (*Person*) which now depends on another module (*Math*), so Cp^3 must be called twice.

Time for g++	Time for Cp^3 & g++	Test

49 secs.	55 secs. (+11%)	`for((i=0;i<100;++i)); do`
		`  ./cp3 Person2.mpp > /dev/null;`
		`  g++ Person2.mpp -o person2;`
		`done`

54 secs.	70 secs (+23%)	`for((i=0;i<100;++i)); do`
		`  ./cp3 Person.mpp > /dev/null;`
		`  ./cp3 Math.mpp > /dev/null;`
		`  g++ Person.cpp Math.cpp -o person;`
		`done`

Table 1: Results obtained in an Intel Core2duo e5200 - 4Gb RAM machine, running Ubuntu GNU Linux 8.04 and using the GNU GCC compiler.

These results are encouraging: for a single module, only little more than a 10% of the time is needed in order to obtain the final executable. When Cp^3 is called twice the extra time needed rises until 23% (under a fourth of the total time), which we think is a very good mark as the growth is just linear.

The Cp^3 tool is not specially designed to decrease compilation times at all; however, there are still opportunities to improve that. Indeed, the preprocessor detects the time stamp of the target

files before blindly reprocessing the module file (which would suppose recompilation of all modules for tools such as *make*).

The specific batch files needed in order to carry out these tests are shown in the third column of the table. Basically, a loop is repeated one hundred times in order to get the statistical results. When measuring only *GNU g++*, the lines beginning with `./cp3` are removed.

8 Conclusions

Module management support for C++ is very poor. Though seasoned programmers are already used, students and beginners find this need of separation of interface and implementation in two parts strange and unnecessary complex.

In this document, the Cp^3 preprocessor has been presented. It does not only eliminate the defects of the file separation, but also enforces good object-oriented programming and procedural programming practices, separating them in practical use, and thus giving a new sense to the hybrid characteristics of C++.

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 file. This is straightforward, easy to understand, easy to use and better for learning C++ (from the perspective of being a "new" language, definitely removing the need of learning C and then C++, as well as the barrier between them).

9 References

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

Vandevoorde, D. (2005). Modules in C++ (revision 2). Std. committee document N1778=05-0138

Vandevoorde, D. (2007). Modules in C++ (revision 5). Std. committee document N2316=06-0176

Hohmuth, M. (2004). Web link: http://os.inf.tu-dresden.de/~hohmuth/prj/preprocess/

Stroustrup, B. (1986). *The C++ Programming Language* . Addison-Wesley.

Stroustrup, B. (1998). An overview of the standard C++ programming language. *Handbook of Object technology*. CRC press LLC Boca Ratón.

10 Appendix A – Test results

Summary

	PRETEST	POSTTEST	Compared
Knowledge of modular programming			
No knowledge	4,76%	19,05%	14,29%
Some	42,86%	66,67%	23,81%
Deep knowledge	52,38%	14,29%	-38,10%
Total	**100,00%**	**100,00%**	**0,00%**
Useful for lecturing			
Yes	95,24%	90,48%	-4,76%
No	0,00%	0,00%	0,00%
I don't know	4,76%	9,52%	4,76%
Total	100,00%	100,00%	0,00%
Module support in C++ is based on			
Separate the program in different files	0,00%	0,00%	0,00%
Use *namespaces*	4,76%	19,05%	14,29%
Divide in modules, and them in .h and .cpp	85,71%	80,95%	-4,76%
I don't know	9,52%	0,00%	-9,52%
None of them	0,00%	0,00%	0,00%
Total	**100,00%**	**100,00%**	**0,00%**
Utility of modular programming			
Useful in theory and practice	76,19%	85,71%	9,52%
Useful for lecturing	0,00%	4,76%	4,76%
Only in theory	4,76%	9,52%	4,76%
I don't know	14,29%	0,00%	-14,29%
No useful at all	4,76%	0,00%	-4,76%
Total	**100,00%**	**100,00%**	**0,00%**
The seminar was useful for you			
Yes	90,48%	85,71%	-4,76%
No	0,00%	14,29%	14,29%
I don't know	9,52%	0,00%	-9,52%
Total	**100,00%**	**100,00%**	**0,00%**
Use a prototype for experimenting is...			
Better than explanations with the blackboard	0,00%	0,00%	0,00%
Better comprehension with a prototype	100,00%	80,95%	-19,05%
It does not matter	0,00%	4,76%	4,76%
It will make you acquire bad habits		4,76%	
Definitely bad		0,00%	
I don't know		9,52%	
Total	**100,00%**	**100,00%**	**-14,29%**

Software use is		
Simple	66,67%	100,00%
Medium	33,33%	
Complex	0,00%	0,00%
Total	**100,00%**	**100,00%**
You don't like...		
Automates	4,76%	33,33%
It is simple	28,57%	
Limits the use of C++	61,90%	61,90%
Nothing	4,76%	4,76%
Total	**100,00%**	**100,00%**
You like...		
Automates	71,43%	85,71%
It is simple	14,29%	
Limits the use of C++	4,76%	14,29%
Nothing	9,52%	
Total	**100,00%**	**100,00%**
Your perception about modular programing		
Changed a lot	9,52%	71,43%
Changed a bit	61,90%	
Didn't changed	28,57%	28,57%
Total	**100,00%**	**100,00%**
Better modular support in C++ would be		
Excellent	38,10%	85,71%
Good	47,62%	
I don't really care	14,29%	14,29%
Total	**100,00%**	**100,00%**
Keep syntax or create a new one		
Keep syntax	90,48%	90,48%
Create new one	4,76%	
I don't know	4,76%	9,52%
Total	**100,00%**	**100,00%**

11 Appendix B – Pretest

PRETEST

Knowledge of modular programming		
No knowledge	1	4,76%
Some	9	42,86%
Deep knowledge	11	52,38%
Total	**21**	
Useful for lecturing programming		
Yes	20	95,24%
No	0	0,00%
I don't know	1	4,76%
Total	**21**	
Module support in C++ is based on		
Separate the program in different files	0	0,00%
Use *namespaces*	1	4,76%
Divide in modules, and them in .h and .cpp	18	85,71%
I don't know	2	9,52%
None of them	0	0,00%
Total	**21**	
Utility of modular programming		
Useful in theory and practice	16	76,19%
Useful for teaching	0	0,00%
Only in theory	1	4,76%
I don't know	3	14,29%
No useful at all	1	4,76%
Total	**21**	
Seminar useful for you		
Yes	19	90,48%
No	0	0,00%
I don't know	2	9,52%
Total	**21**	
Use a prototype for experimenting is...		
Better than exercises in the blackboard	0	0,00%
Better comprehension with a prototype	21	100,00%
It does not matter	0	0,00%
Total	**21**	

12 Appendix C – Posttest

POSTTEST

Knowledge of modular programming		
No knowledge	4	19,05%
Some	14	66,67%
Deep knowledge	3	14,29%
Total	**21**	
Useful for lecturing		
Yes	19	90,48%
No	0	0,00%
I don't know	2	9,52%
Total	**21**	
Module support in C++ is based on		
Separate the program in different files	0	0,00%
Use *namespaces*	4	19,05%
Divide in modules, and them in .h and .cpp	17	80,95%
I don't know	0	0,00%
None of them	0	0,00%
Total	**21**	
You think that your knowledge about modules now is		
Low	4	19,05%
Avergae	13	61,90%
High	4	19,05%
Total	**21**	
The seminar was useful for you		
Yes	18	85,71%
No	3	14,29%
I don't know	0	0,00%
Total	**21**	
Your comprehension improved		
Yes	19	90,48%
No	2	9,52%
I don't know	0	0,00%
Total	**21**	

Use a prototype for experimenting is...		
Better than exercises in the blackboard	0	0,00%
Better comprehension with a prototype	17	80,95%
It does not matter	1	4,76%
It will make you acquire bad habits	1	4,76%
Definitely bad	0	0,00%
I don't know	2	9,52%
Total	**21**	
Software use is		
Simple	14	66,67%
Medium	7	33,33%
Complex	0	0,00%
Total	**21**	
You like...		
Automates	15	71,43%
It is simple	3	14,29%
Limits the use of C++	1	4,76%
Nothing	2	9,52%
Total	**21**	
You don't like...		
Automates	1	4,76%
It is simple	6	28,57%
Limits the use of C++	13	61,90%
Nothing	1	4,76%
Total	**21**	
Your perception about modular programing		
Changed a lot	2	9,52%
Changed a bit	13	61,90%
Didn't changed	6	28,57%
Total	**21**	
Better modular support in C++ would be		
Excellent	8	38,10%
Good	10	47,62%
I don't really care	3	14,29%
Total	**21**	
Utility of modular programming		
Useful in theory and practice	18	85,71%
Useful for teaching	1	4,76%
Only in theory	2	9,52%
I don't know	0	0,00%
No useful at all	0	0,00%
Total	**21**	

Keep syntax or create a new one		
Keep syntax		
Create new one	19	90,48%
I don't know	1	4,76%
Total	1	4,76%
	21	