IMPROVING ECONOMIC AND TECHNICAL EFFICIENCY OF PROCEDURAL PROGRAMMING IN COMPILABLE, STATICALLY TYPED C-LIKE LANGUAGES ACCORDING TO THE SPEED OF THE PROCESS AND THE STABILITY OF THE PROGRAM IMPLEMENTATION

Arman MARTIROSYAN
YSU, Doctor of Economic Sciences
Martin MIRZOYAN
UESTC, CSE School, master's student
Vahagn GISHYAN
NPUA, ITTE Institute, SAED, master's student
Yerevan, Armenia

Key words: statically typed, compilable, C-like programming languages; procedural programming; functions; optimization.

Introduction. Currently, there is a trend in the field of programming towards compiled and high-performance languages. There is a competition among companies listed in international indexes and universities with global scientific qualifications. They are trying to create a new technology that will correct the main problems in C-like languages [5][7][8][9]. This newly created technology will make it possible to make the products of these organizations in the market more competitive, more controllable, more flexible, and more ready for daily challenges.

C-like languages are being considered to develop such technology, the reason being that it is currently impossible to imagine any programming language that has not been influenced by C [1][6]. It is the basis of such languages as: C++, Java, C#, JavaScript, Python, PHP, etc. Furthermore, several languages that were developed before C or that did not originally consider its syntactic and semantic features acquired extensions that resembled C later. They are Shell, SQL, etc.

C language is based on functions [4], which are the main components of software implementation. However, when designing the C language, the compilers were still not powerful enough [3], and compilation stage checks, decisions, improvements were almost not assumed in the language. The situation changed somewhat after the design of the C++ language, which introduced compile-time overrides. However, at the very beginning of the language's design, in the 80s, compilers were not powerful enough either. Even today, there are compile-time improvements that are not present in the language.

This article presents several recommendations aimed to make the procedural programming components of C-like languages [2] safer and increasing their speed at run-time through compile-time checks/decisions.

Methodology. The theoretical basis of this scientific work is the speeches made at international conferences, research of universities with world scientific qualifications, attempts to replace C-like languages of companies listed in international indexes, scientific works, annual reports published by standardization committees, etc. The problem is that the current market is undergoing changes with great speed and in this context the flexibility of software implementations is important. For this purpose, the article proposed innovations that aim to facilitate the process of software implementation, readability, and further expansion.

Literature review. Currently, companies are trying to solve the problem as such as Google with Go and Carbon languages, Microsoft with cpp2 language, Mozilla and Linux with Rust, etc. There are also individual attempts to solve the problem, for example with D or Odin. The most successful and popular technologies are Go and Rust. These make the software product more secure, more competitive and contains fewer bugs and is implemented in a shorter period. The US National Security Service (NSA) has even advised not to use C/C++ programs, because these languages are vulnerable from the point of view of malicious attacks. Instead, it is recommended to use C#, Java, Go, Rust, Ruby languages [10].

Even though such above-mentioned languages make software products more competitive and more secure, C and C++ languages do not become less useful. The reason lies in the fact that the latter provide more speed, are more suitable for developing mathematical models and provide an opportunity to perform calculations at the compilation stage, to have a higher performance at the run-time.

This article offers recommendations for compilable, statically typed, C-like procedural programming languages that will make the products implemented more competitive in terms of quality, processing speed, and security.

Scientific novelty. Constancy of functions. There is a rule in the C++ programming language: when any memory range is declared constant (const) at runtime, it cannot be changed implicitly (const_cast, mutable). Any non-obvious attempt to modify the memory will result in a compile-time error. A different rule is proposed. all memory ranges that are declared constant cannot be modified under any conditions at compile or run-time. It is now possible to declare a function as a constant. This means that the function does not modify the state or environment of any object with the global access. For example, it does not open any file and does not write to it. To declare a function constant, write const after the name.

Example:

```
const std::string  doSomething(const std::string&)   const { … }
void               swap (int&, int&)                  const { … }
```

Only a constant function can be called inside a constant function. Constant function has lots of opportunities for improvement. If a value was passed to the constant function which was known at the compilation stage, then the latter can be determined at the compilation stage itself. In other words, the constant function has no side operations, but only a counting role.

Conditions of values. These are mandatory checked, compile-time rules that, if violated, will result in a compile-time error. For example, when a function receives a pointer and we want to declare that the pointer cannot be nullptr. At compile time, checks are made to pass pointers that meet the condition to the function. Example:

```
int foo(const int* ptr != null)              const { … }
int foo(const int index != -1)               const { … }
int foo(const int index != { 0,  1, 11 })    const { … }
```

The condition can be a constant function.

Ranges of values. Given, that we have an abstraction called *String* that regulates work with an array of symbols. It is necessary to write an indexing function that receives an integer, index, and returns the corresponding symbol. The latter has two possible implementations, with and without condition checking.

The option of conditional check is safe, but an additional check is performed (which is time-consuming) for all cases where it is possible to guarantee at compile time that an argument satisfies the conditions.

There is no extra waste of time in the no-condition-check option, but an incorrect argument leads to unpredictable behavior even with compile-time error discovery guarantees.

To solve the above problem, compile-time checking of the function parameter "range of values" is used, which is a special case of the value condition. Example:

```
const char get (const int index { >= 0, <  line.size() })    const {…} // 0
const char get (const int index { <  0, >= line.size() })    const {…} // 1
const char get (const int index)                             const {…} // 2
```

Thus, the function gets three implementations. The first is for all cases where at compile time one can guarantee *index* to be in the range of *[0; line.size())*. Second, for all those cases where at compile time it is possible to guarantee *index* not to be in the range of *[0; line.size())*. And thirdly, when nothing can be guaranteed at the compile time.

Ignoring the return value. Programming languages such as C typically do not strictly approach to ignorance to the memory scopes. The return value of a function can

be ignored, and there is no way to prevent it. This is more sensitive for constant functions.

Nevertheless, there are cases when the return value of a non-constant function is an event. To solve this problem, it is recommended to enable function overloading according to the return value. Example:

bool dosomething(int& value) {…}
void dosomething(int& value) {…}

There can be a single overload of the function according to the return type, or return nothing.

1. Access modifiers:

By declaring the function *public*, we indicate that it is visible from other compilation modules (static and dynamic libraries, executable output file).

By declaring the function *protected*, we indicate that it is visible only from that module.

By declaring a function *private*, we indicate that it is visible only within the scope of that file.

Orientation by namespace. The global space is only one, which causes a name conflict problem. The reason for the absence of namespaces in the C language until today is the preservation of predictability at the level of the Assembly language. On the other hand, the presence and competent use of namespaces brings the readability of the program implementation and the ease of further extensions. For this purpose, it is proposed:

1. Block the use of global spaces in software implementation. All components of a software implementation must reside in a namespace.
2. Strict the file structure of the software implementation. The namespace asserts the fact that the source files are located in the directory of the same name or have the file name. This means that there is only one way to expand that namespace: to be located in the certain directory.

Examples. *«greater-int» program:*

```
namespace greater-int
{
    int greater(const int v1, const int v2) const
    {
    return (v1 > v2 ? v1 : v2);
    }
    int main()
    {
```

```
        int value1, value2;
        std::cin >> value1 >> value2;
        const int valueg = greater(value1, value2);
        std::cout << "greater is " << valueg << std::endl;
        return (0);
        }
}
```

«index-of» program:

```
namespace index-of
{
        //for checked arr
        int index-of(const char* arr != nullptr, const char symbol) const
        {
                const int index = 0;
                while (arr[index] != '\0')
                {
                        if (arr[index] == symbol)
                        {
                        return (index);
                        }
                        ++index;
                        }
                        return (-1); // not found
                        }
        //for unchecked arr
        int index-of(const char* arr, const char symbol) const
        {
                         if (arr == nullptr) // incorrect input
                         {
                         return (-1);
                         }
                         return (index-of(arr, symbol)); // arr is already checked
        }
        int main()
        {
                    bool userinput;
                    std::cin << userinput;

                    const char* arr = nullptr;
                    if (userinput == true)
                    {
                    arr = "some text";
                    }
                    const char symbol = 'e';
                    const int index index-of(arr, symbol);
```

```
            if (index != -1)
            {
            std::cout << "index is "<< index << std::endl;
            }
            else
            {
            std::cout << "error, function return -1" << std::endl;
            }
            return (0);
      }
}
```

«user-age» program:

```
namespace user-age
{
bool input-number(int& input) const
{
std::string str-input = "";
std::cin >> str-input;
if (str-input.is-not-number())
{
return false;
}
input = str-input.to-int();
return true;
}
void input-number(int& input) const
{
std::string str-input = "";
std::cin >> str-input;
if (str-input.is-not-number())
{
return;
}
input = str-input.atoi();
}
int main()
{
{
int user-age;
const bool result input-number(user-age);
if (result != true || user-age < 0)
{
std::cout << "input is not correct" << std::endl;
}
else
```

```
{
std::cout << "user age is " << result << std::endl;
}
}
{
int user-age = -1;
input-number(user-age);
if (user-age < 0)
{
std::cout << "input is not correct" << std::endl;
}
else
{
std::cout << "user age is " << result << std::endl;
}
}
return (0);
}
}
```

Analysis. The IT sector is constantly growing. Today, companies whose processes are not automated and are not available in the global network are not considered competitive. The problem lies in the fact that the implementation of the actual IT product is often neglected here. It can be competitive at some point and even have real growth, but how successful the product is depends on product risk, speed of launch, flexibility to implement new changes, stability, cross-platform capability, and protection against attacks. The only way to develop a more competitive product is to develop a language that meets these standards.

The main feature of IT products is the potential for endless updates and extensions. However, the product can be developed in such a way that adding new changes can be many times more expensive from a financial point of view than creating a new product from scratch. This leads to the fact that competing companies that initially implemented the more flexible product become more competitive and more likely to achieve greater success in the long run.

The presented changes make the software more flexible for changes, thus making the product more competitive not only today, but also for the future. Software implementation becomes easier to comprehend and handle making the code cleaner.

Clean code is code that is easy to understand and easy to change. It is easy to understand the execution flow of the entire application, how the different objects collaborate with each other, the role and responsibility of each class, what each method does, and finally what is the purpose of each expression and variable [11].

Implementing quality code leads to a reduction in the number of bugs in the product, and making changes requires less time and resources. Making changes to unreadable and unintelligible code is more expensive from a financial point of view, and sometimes it is more appropriate to re-implement it wholly.

The changes proposed in the article improve the implementation of software products, make the products themselves more competitive. However, in contrast to the improvements, there are also shortcomings. Improvements have been made at the compile time stage, slowing it down. Experience shows that the end user, who will compile once at worst, can put up with the slowness of the latter, but complains when the executable phase is slow. As a result of the changes, the execution phase is accelerated at the expense of the compilation phase.

One of the disadvantages is that the time required at the initial stage of the software implementation of the product has also increased. This is compensated by the fact that it takes less time to add changes later.

Conclusion. In this work, proposals for compilable, statically typed, and C-like languages were presented which can make procedural programming in these languages, i.e., work with functions, safer, readable, and increase the speed of the process. These criteria express the competitiveness of products. Products developed in change-aware languages have increased security, development speed, and flexibility. These results are achieved through compile-time checks and decisions. Change-aware languages can be used in software development of the banking, government, judicial and free market sectors.

Presented changes can be included in arbitrary compilable, statically typed programming languages which also apply functions/methods. This applies both to existing languages and newly created ones. They make products more flexible to changes and more competitive in the market.

References

1. https://www.opensourceforu.com/2011/11/joy-of-programming-legacy-of-c/
2. https://en.wikipedia.org/wiki/Procedural_programming
3. https://cmustdie.com/
4. "Welcome to IEEE Xplore 2.0: Use of procedural programming languages for controlling production systems". *Proceedings. The Seventh IEEE Conference on Artificial Intelligence Application*.
ieeexplore.ieee.org. doi:10.1109/CAIA.1991.120848. S2CID 58175293
5. https://github.com/carbon-language/carbon-lang

6. Brajan Kernigan, Dennis Ritchi. Jazyk programmirovanija C. Moskva: Vil'jams, 2015. 304 s. ISBN 978-5-8459-1975-5.
7. https://herbsutter.com/2022/09/19/my-cppcon-2022-talk-is-online-can-c-be-10x-simpler-safer/
8. Klabnik, S., & Nichols, C. (2019). *The Rust Programming Language (Covers Rust 2018)*. No Starch Press.
9. Matthews, B. (March 10, 2019). *Programming in Rust: the good, the bad, the ugly*. Hackernoon.
10. National Security Agency | Cybersecurity Information Sheet, U/OO/219936-22 | PP-22-1723 | NOV 2022 Ver. 1.0
11. Martin, R. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship.* Pearson.

Arman MARTIROSYAN, Martin MIRZOYAN, Vahagn GISHYAN

Improving economic and technical efficiency of procedural programming in compilable, statically typed C-like languages according to the speed of the process and the stability of the program implementation

Key words: statically typed, compilable, C-like programming languages, procedural programming, functions; optimization

In the modern IT industry, there is a trend towards languages with high performance. The products implemented through them become more efficient and energy-saving. However, these languages have a number of disadvantages based on their technical implementation. These defects create side problems over time and increase the cost of implementing, securing, configuring, and extending the functionality of the product. Thus, the product becomes more expensive from a financial point of view, but more competitive. In order to further increase the competitiveness of products, to neutralize side problems, to reduce the cost, companies listed in international indexes and universities with global scientific qualifications either develop a new language, or update and add new functionality to the already existing language. The article presents recommendations for compilable, statically typed, C-like languages to make their procedural programming safer. They will ensure readability and increase the speed of the process. These changes will make IT products more competitive in the free market. The advantage of recommendations is that the most time-consuming checks and decisions will be made at compile time. There are some examples of the changes in the article.