Technology & Management Review **Vol 1, No 1 (2016)**

The Power and Legacy of C Programming: A Deep Dive into the Language

[1]**Parikshith Reddy Baddam,** *Software Developer, Data Systems Integration Group, Inc., USA*
[2]**Swathi Kaluvakuri,** *Sr. Software Engineer, Quantum Technologies, Hyderabad, INDIA*

Abstract:
The 1970s-born C programming language is still used in computer science for its efficiency, portability, and impact on software development. We show how C's simplicity hides its enormous influence by examining its elegant syntax and low-level manipulation capabilities. The language is ideal for systems and embedded programming because of its unequaled control and expressiveness. C's core ideas, historical context, and ongoing influence are examined in this article. The article discusses C's role in milestone projects like Unix and its mission-critical performance. Real-world case studies demonstrate C's relevance and adaptability to modern computing demands, solving security concerns with C11. This exploration of C programming's power and history reveals a language that guides software development decades later. Understanding C is more than a historical pursuit—it unlocks modern software ideas.

Keywords: C Programming Language, Software Development, Modular Programming, Control Flow, Memory Management

INTRODUCTION

The C programming language is one of the few that has been able to withstand the passage of time, and the advancement of technology to the same degree as other programming languages are constantly changing. C developed as a revolutionary force during the early 1970s at Bell Labs, where it was forged into a programming language that would surpass its initial modest objectives and influence the very fabric of computer science and software development. C was born in the furnace of Bell Labs. This article goes on a comprehensive investigation of the strength and long legacy of C programming. It does so by digging into the language's core concepts, historical evolution, and ongoing impact on the ever-expanding domain of technological advancement.

C, at its very essence, embodies both simplicity and elegance in equal measure. Its syntax, despite giving the impression of being straightforward, conceals a depth of talent that has reverberated down the decades. Programmers looking for a language that strikes a good balance between expressiveness and performance frequently turn to C because of its adaptability, which allows it to be used for everything from low-level systems programming to the complexities of embedded

system programming. This essay will attempt to decipher the complexities of C's design philosophy by analyzing its founder, Dennis Ritchie when developing the programming language and shedding light on how those choices have led to C's tenacity and continuous relevance (Xiao, 2013).

C is the progenitor of a family of programming languages, each of which owes a debt to the pioneering notions that C introduced. This significance extends beyond C's immediate value. As we move through C's family links to related programming languages such as C++, Objective-C, and C#, we become aware of the significant impact that C has had on the development of contemporary software engineering. It is not only a matter of syntax; C has bequeathed underlying principles that support object-oriented programming. Concurrent with its role in influencing the course of computer science education and industry practices, C has also played a significant part in shaping the industry's practices.

The tale of C's history is intricately entwined with the evolution of illustrious software systems, none of which are more celebrated than the Unix operating system. Case studies based on the natural world shed light on how the efficacy and performance of C became the foundation of mission-critical initiatives, further solidifying C's status as the language of choice for endeavors that require precision and reliability.

C has not been a company that has stood still in the face of a shifting technology landscape. This article delves into the modern modifications of the C programming language and investigates how it has adapted to meet the demands of modern computing. Incorporating standards such as C11 and a proactive approach taken by the language towards security concerns demonstrate a commitment to evolution without abandoning the ideas upon which it was founded.

Our objective, as we begin this in-depth investigation into the power and legacy of C programming, is not only to investigate a programming language's history but also to comprehend the fundamental tenets that continue to influence the structure of contemporary software. Programming in C is more than just a historical artifact; it is a living witness to the ongoing impact that innovative design, simplicity, and adaptability in programming languages have had and continues to have.

FOUNDATIONS OF C

Dennis Ritchie developed the computer programming language C at Bell Labs in the early 1970s. It is regarded as one of computer science's most essential building blocks. Its ongoing popularity and effect may be traced back to a set of core concepts that have withstood the test of time despite the onslaught of technological advancement. This investigation into the foundations of C digs into its fundamental principles, analyzing how its design choices and core characteristics have formed its trajectory and influenced the larger landscape of programming languages. Specifically, this study looks at how C has influenced the evolution of the C programming language (Yassine et al., 2017).

Simplicity and Elegance: C is defined by its central focus on understated elegance and uncomplicated beauty. Programmers can express complex concepts with clarity and accuracy thanks to the language's syntax, which gives the appearance of being overly simple. C's brevity has shown to be extremely useful, particularly in fields where resource efficiency and performance are of the utmost importance, in contrast to more verbose programming languages. This dedication to keeping things straightforward makes it easier to pick up new skills and helps to produce reliable and effective code.

Low-Level Manipulation: Programmers are given control over the computer's hardware resources unrivaled by many other languages. This level of control is provided by C.C., which allows developers to interface directly with the underlying hardware thanks to its support for pointers and manual memory management, which is a programming language. This flexibility to manipulate at a low level has proven extremely useful in creating operating systems, embedded systems, and other performance-critical applications. On the other hand, C is a language that rewards programming expertise and practices that are extremely careful since it requires a heightened awareness of the system's complexities.

Portability: The fact that C places such a premium on portability is one of the primary reasons for its widespread adoption. A program written in C can be produced and run on several platforms with minor adjustments. This portability is made possible by the ANSI C standard, which standardized the characteristics of the language and assured that other implementations of the language behaved in the same way. This property has proven particularly useful in creating cross-platform programs and systems, contributing to the widespread adoption and continued use of the C programming language.

Procedural Paradigm: The C programming language adheres to the procedural programming paradigm, which emphasizes the sequential execution of functions and procedures. This methodology encourages a clean and modular code structure, making it an excellent choice for various project sizes. The procedural paradigm, when combined with the straightforward nature of C, supports the development of codebases that are both effective and easy to maintain. Although later languages have incorporated more complicated paradigms, the procedural underpinning of C is still widely used and serves as the cornerstone for various software engineering processes.

Influence on Modern Languages: C's foundations reach far beyond the syntax and features of the language itself and have considerably impacted the development of many contemporary programming languages. C++, a direct descendant, was the first programming language to introduce object-oriented programming (OOP) principles and elegantly integrated those concepts into the existing procedural framework. Another progeny, Objective-C, was a significant contributor to the application development process for both macOS and iOS. Furthermore, languages such as C# owe their existence to the foundations laid by C, particularly in the context of the Microsoft.NET framework. C was the language that laid the foundation for these languages. Learning C gives one a profound insight into the ancestry of contemporary programming languages, revealing how fundamental concepts continue to be used and adapted across generations of computer languages.

Role in System Development: Building dependable and effective software systems is firmly rooted in the fundamentals of the C programming language. Language was an essential component in creating the Unix operating system, considered a lynchpin in the annals of

computer history. C became the language of choice for Unix because of its ease of use, high performance, and portability. This contributed to the widespread acceptance of the operating system and the programming language in academic and industrial settings. Real-world case studies of system development highlight C's capacity to address complex difficulties, showcasing the language's resiliency and usefulness in essential applications.

Adaptability and Standards: The fundamentals of C do not relegate it to the annals of history; instead, the programming language has adapted to the shifting landscape of computers to meet the needs of its users. Thanks to the establishment of standards, such as the widely used C89, and subsequent updates, such as the C11, the C programming language can stay relevant and adapt to the ever-changing requirements of software developers. These standards provide a road map for the evolution of the language by addressing difficulties, providing new features, and keeping compatibility while at the same time respecting the fundamental concepts that characterize C.

KEY FEATURES

The C programming language has maintained its widespread use and influence for several decades thanks to fundamental characteristics that have weathered the test of time. Since its introduction at Bell Labs in the early 1970s, C's architectural choices and features have laid the groundwork for efficient, portable, and versatile programming. Its continuous significance in contemporary software development is a testament to this. This investigation dissects the fundamental characteristics that set apart C and contribute to its enduring impact.

Simplicity and Efficiency: A commitment to simplicity and efficiency lies at the heart of the C programming language. Because the language's syntax is condensed and straightforward, programmers can express complicated ideas with minimal ambiguity. This simplicity allows for a short learning curve, which makes C accessible to novices while still enabling more experienced developers to build code that is both efficient and elegant. The minimalistic style of the language is a good fit for its position in systems programming, where resource efficiency is of the utmost importance.

Portability: The portability of C is one of its most notable characteristics. Code written in C is portable and can be quickly adapted to run on various platforms with minor adjustments. This portability is based on the ANSI C standard, which standardized the language and ensured a uniform behavior across many implementations. ANSI C is the foundation for this portability. C is a flexible option for projects that require interoperability across various operating systems and hardware architectures because of this feature's significant contribution to the development of cross-platform applications.

Low-Level Manipulation: C's capabilities, such as pointers and manual memory management, provide programmers with control over the resources available to their computer's hardware. This feature of low-level manipulation is handy when having precise control over system resources is necessary, such as when operating system development or embedded system programming is being done. On the other hand, it necessitates a sophisticated comprehension of the system's architecture and rigorous programming approaches.

Procedural Programming Paradigm: The emphasis on organizing code into functions and procedures is central to the C programming paradigm, known as procedural programming. This paradigm emphasizes the use of modular design, which makes it easier to write code that is both readable and manageable. Considering each function as a separate, self-contained unit contributing to the program's overarching structure is possible. The procedural paradigm, which involves the execution of functions in a step-by-step fashion, is compatible with the fundamental ideas that underpin C, and it has had an impact on later programming languages (Hong & Li, 2013).

Extensive Standard Library: C comes with a comprehensive standard library containing various functions that can be used for routine activities. This library features routines for input and output, manipulating strings, allocating memory, and more. Programmers can take advantage of pre-built functions for frequently used tasks thanks to the availability of a complete standard library, which makes the development process much easier and eliminates the need to begin from scratch. C programmers should expect increased efficiency and productivity due to this functionality.

Influential on Contemporary Languages: C's influence extends far beyond the scope of its codebase, as seen in how many modern programming languages are structured. For example, C++ expands upon the fundamentals established by C and presents principles of object-oriented programming. The development of applications for macOS and iOS relies heavily on Objective-C, which was initially conceived as an extension of the programming language C. The concepts established by C have entered the DNA of other languages, such as C# and Java, which has impacted the design choices that later generations of computer languages have made.

Adaptability through Standards: The adaptability of C is demonstrated by the fact that it strictly adheres to standards, which helps to maintain a coherent and developing language. The language has incorporated new features, addressed issues, and held compatibility thanks to establishing standards such as ANSI C (C89) and subsequent updates such as C11. The dedication to standards guarantees that C will continue to be an up-to-date and viable option for developers tasked with negotiating the complexity of modern software development.

THE ANATOMY OF C PROGRAMS

C programming is simple and efficient, yet its anatomy gives it power. C programs are designed to be precise, portable, and versatile as a foundational language. This study examines C programs' key components, which make C essential to software development.

Header Files: Header files are often the first step in C program anatomy. These ".h" files declare functions, macros, and other essentials. They guide the compiler through the program's structure and interfaces. Header files make code modular and easy to handle.

Primary Function: Every C program has a primary function. The primary function contains the program's logic and starts execution. Declare variables, invoke functions, and execute statements sequentially. The primary function's simplicity and clarity demonstrate C's procedural programming (Sui et al., 2014).

Variables and Data Types: C's anatomy carefully handles variables and data types. Variable types and names are declared at the start of functions or blocks to allocate memory. C supports basic data types like int, float, and char, letting programmers choose the best representation. Explicit data typing improves code reliability and efficiency.

Functions: C programs use functions to encapsulate tasks and promote modularity. C allows developers to add functions beyond the primary function, promoting code reuse and maintainability. Function prototypes, usually stated in header files, preview the function's structure and let software components communicate.

Control Flow Statements: Control flow statements—if, else, loops—enhance C's anatomy. Developers can design dynamic logic using these statements to control the program's execution route. Complex program logic is expressed efficiently in C's control flow due to its simplicity and transparency.

Pointers and Memory Management: C's anatomy allows low-level memory address manipulation via pointers. Pointers improve the language's data structure and memory interaction. Malloc and free allow developers to dynamically acquire and deallocate memory, improving C's resource efficiency.

Arrays and Strings: C programs use arrays and strings to organize and manipulate data. The array's fixed size matches C's focus on memory efficiency and predictability. C's string manipulation using character arrays shows its text data handling capabilities.

Standard Input/ Output (I/O): The strong printf and scanf utilities standardize input and output in C. These functions facilitate program-user or external system interaction, improving the language's versatility.

Preprocessor Directives: C programs' anatomy is shaped by "#" preprocessor directives. They tell the preprocessor to include header files, define constants, and conditionally compile code. These directives help C code adapt and configure.

Comments: Comments are an essential part of C's anatomy that are often overlooked. Single-line "//" comments and multi-line "/* */" comments give context and documentation. They improve software comprehension, debugging, and developer collaboration.

DATA TYPES

When it comes to the world of computer programming, the data types serve as the foundation upon which the efficiency and precision of the C language are created. It is necessary to have a solid understanding of the complexities of data types to develop resilient, memory-efficient, and high-performance programs. This investigation digs into the varied universe of data types in C, shedding light on the significance of these types and how they contribute to language adaptability (Pei et al., 2014).

Basic Data Types: The programming language C supports a collection of fundamental data types designed to represent a particular value. These are the following:

- **int:** The notation int represents integers, usually whole numbers.
- **float:** The float data type represents integers with a single-precision floating-point format compatible with decimal values.

- **double:** The notation double represents numbers with a floating-point precision of two, providing a higher level of precision than the float notation.
- **char:** is short for "character," and stands for a single character.

Derived Data Types: The language gains both abstraction and versatility because of the use of derived data types formed from the basic data types. The following are important examples:

- **Arrays:** Arrays are collections of elements that have the same data type. This allows for the efficient storage and retrieval of values that are related to one another.
- **Structures** are aggregate data types that make it possible to combine several different data types under a single name. This makes the building of complicated structures much simpler.
- **Pointers:** Pointers are variables that record memory addresses and make it possible to manipulate and dynamically allot memory space directly.

Enumeration Data Type: C's enumeration data types make it possible to define a set of named integral constants, which significantly improves the readability of the program's source code. Enumerations enhance the readability of the code by giving meaningful names to the integer values they represent, making the code easier to maintain.

Void Data Type: When declaring a value is unnecessary, the null data type is used as a stand-in for other data types. For instance, the fact that a function has a void return type indicates that it does not return any value when the function is called.

Size and Precision Modifiers: C permits the change of fundamental data types by applying size and precision modifiers, making it possible to support various value ranges. Take, for example:

- **short:** Changes the representation of int so that it can handle lower integer values.
- **long:** Adjusts values represented by int or double to be more extensive or precise.

Signed and Unsigned Modifiers: The behavior of the fundamental data types can be further refined using signed and unsigned modifiers. Integers are signed by default and may store positive and negative values. By limiting the range to only non-negative numbers, the unsigned modifier practically doubles the amount of space available for positive values (Bošanský & Patzak, 2016).

Typedef: The typedef keyword allows users to create user-defined data types and provides a mechanism for aliasing existing data types with more informative names. This makes the code easier to read and helps develop a better grasp of data structures.

Constants: In C, constants are values that cannot be changed and always store the same value. These constants, whether numerical, symbolic, or character, contribute to forming fixed and unchanging values inside the program.

Standard Library Data Types: Header files such as stdint.h and stddef.h, part of the C standard library, is responsible for introducing new data types. Fixed-width integer types, such as int8_t and uint16_t, as well as size-related types, such as size_t, are among those that contribute to portability and precision across a variety of systems.

CONTROL FLOW

Control flow in the C programming language determines how statements are carried out. This gives developers the ability to construct programs that are dynamic, responsive, and efficient. Control flow structures allow programmers to modify the flow of execution of their code by using techniques such as conditional branching and looping expressions. This investigation goes deep into the fundamentals of control flow in C, illuminating the underlying principles that determine how a program moves through the many stages of its logic.

Conditional Statements: The evaluation of certain conditions can trigger the execution of predetermined code sections through conditional statements. The following are the most common conditional statements used in C:

- **if Statement:** The if Statement is a conditional statement that determines whether or not to execute a piece of code depending on the outcome of a test.

```
if (condition) {
   // Code block to run if the condition is true
}
```

- **else Statement:** offers an alternate chunk of code that will be carried out if the condition contained in the matching Statement is not met.

```
if (condition) {
   // Code block to execute if the condition is true
} else {
   // Code block to execute if the condition is false
}
```

- **else Statement:** enables the evaluation of various conditions to be carried out step-by-step.

```
 if (condition1) {
   // Code block to execute if condition1 is true
} else if (condition2) {
   // Code block to execute if condition2 is true
} else {
   // Code block to execute if none of the conditions are true
}
```

Switch-Case Statements: Handling several situations clearly and succinctly is made possible by the switch-case Statement. It takes an expression, evaluates it, and then runs the code block corresponding to the case it matches (Chrysafiadi & Virvou, 2013).

```
switch (expression) {
   case constant1:
      // Code block to execute if expression matches constant1
      break;
   case constant2:
```

```
        // Code block to execute if expression matches constant2
        break;
    // ... additional cases
    default:
        // Code block to execute if no case matches the expression
}
```

Looping Constructs: Loops allow for the repetitive execution of a block of code, which enables fast iteration across data structures or until a particular condition is fulfilled. Loops can also be used to check if a specific condition is met. The following are examples of popular looping constructs in C:

- **for Loop:** Will carry out an assignment of code a predetermined number of times via the Loop.

```
for (initialization; condition; update) {
    // Code block to execute during each iteration
}
```

- **while Loop** is a type of Loop used to execute a block of code for as long as a specific condition remains true.

```
while (condition) {
    // Code block to run while the state is true
}
```

- **do-while Loop:** is analogous to a while loop, except that the code block will be carried out at least once before the condition is evaluated.

```
do {
    // Code block to execute at least once
} while (condition);
```

Jump Statements: Jump statements are available in C, enabling programmers to deviate from the typical control flow of an application. The following are essential jump statements:

- **break:** Using the break keyword will cause the current iteration of a loop or switch-case expression to be terminated.
- **continue:** The continue instruction bypasses the remainder of the loop body and advances to the following iteration.
- **goto:** Control is then passed on to a statement in the program that has been tagged.

Conditional Operator (Ternary Operator): Expressions can be written more compactly by using the conditional operator ('?:'), which allows for the formulation of conditional statements (Lopes et al., 2017).

```
result = (condition) ? value_if_true : value_if_false;
```

This operator evaluates the condition; if it is determined that the condition is satisfied, the value 'value_if_true' is assigned to the result; otherwise, the value 'value_if_false' is assigned.

MEMORY MANAGEMENT IN C

Allocating and freeing memory for variables and data structures is essential in C programming. C is a low-level programming language that directly controls memory, giving exceptional efficiency and versatility. This article discusses C memory management, including allocation, deallocation, and manual memory handling risks.

Stack and Heap: C memory has two regions: the stack and the heap.

- **Stack:** Function call management and local variable storage occur on the stack. Last-in, first-out (LIFO) allocation, and deallocation occur automatically as functions call and return.
- **Heap:** The heap is a dynamic memory pool with manual allocation and deallocation. Heap memory provides extended and controlled data structure storage until expressly removed.

Dynamic Memory Allocation: C provides dynamic heap memory allocation functions. The main functions for this are `malloc`, `calloc`, `realloc`, and `free`:

- **malloc:** Malloc allocates several bytes of memory.
  ```
  int *ptr = (int *)malloc(10 * sizeof(int));
  ```
- **calloc:** Calloc allocates a set number of bytes per block memory block.
  ```
  int *ptr = (int *)calloc(10, sizeof(int));
  ```
- **realloc:** Changes the size of the previously allocated memory block.
  ```
  int *ptr = (int *)malloc(10 * sizeof(int));
  ptr = (int *)realloc(ptr, 20 * sizeof(int));
  ```
- **free:** Frees memory.
  ```
  free(ptr);
  ```

Dynamic memory allocation is essential for handling data structures of different sizes and reducing memory usage.

Memory Leak: Memory leaks are a problem with manual memory management. Memory leaks occur when allocated memory is not correctly deallocated, gradually exhausting memory. C programmers must carefully release dynamically allocated memory to avoid leaks (Bergel et al., 2011).

Dangling Pointers: Dangling pointers point to deallocated memory due to improper pointer handling. Undefined behavior and program crashes can come from dereferencing dangling pointers. C programmers must manage pointers carefully to avoid these difficulties.

Memory Alignment: Optimizing memory access and performance requires alignment. Most C compilers align data types automatically. For applications that require exact memory layout management, programmers may use compiler-specific directives or pragma statements to impose alignment constraints.

Automatic Variables: The compiler automatically manages stack-stored function variables. These variables' memory is automatically deallocated after the function terminates. This

automatic memory management simplifies local variable handling but limits scope and lifetime.

Memory Copy Operations: C has standard functions like `memcpy` and `memmove` for memory copying. Effective data manipulation requires understanding these functions.

Memory Mapping: Memory mapping lets C programs treat files like memory arrays. Functions like `mmap` enable efficient file I/O by mapping files into memory.

FUNCTIONS AND MODULAR PROGRAMMING

C modular programming relies on functions to break down significant issues into manageable and reusable components. This method improves code organization, reusability, maintainability, and readability. This investigation of C functions and modular programming reveals ways for efficient and scalable solutions.

Function Declaration and Definition: A C function is declared at the program's start or in a header file and defined later in the code. Function declarations contain return types, names, and parameter types. Implementation comes from the function definition.

```
// Function declaration
int add(int a, int b);

// Function definition
int add(int a, int b) {
   return a + b;
}
```

Function Prototypes: As forward declarations, function prototypes tell the compiler a function's signature before implementation. To enable modular programming, header files include prototypes.

```
// Function prototype in a header file
int add(int a, int b);
```

Parameters and Return Values: C functions can take parameters and return values, allowing data flow. We can pass parameters by value or reference using pointers.

```
int add(int a, int b) {
   return a + b;
}
```

Scope and Lifetime: Function variables have local scope and are usually created on the stack. They only exist during function execution and are automatically deallocated when it terminates. Encapsulation prevents naming conflicts and maintains data integrity.

Recursive Functions: C allows recursive functions to call themselves. Recursive functions solve issues with comparable sub problems well (Yordzhev, 2013).

```
int factorial(int n) {
   if (n <= 1) {
      return 1;
   }
```

```
    return n * factorial(n - 1);
}
```

Modular Programming: Modular programming breaks a program into independent modules or functions. Each module handles a single task, streamlining code reuse and maintenance.

```
// Module 1
int add(int a, int b) {
    return a + b;
}

// Module 2
int subtract(int a, int b) {
    return a - b;
}
```

Header Files and Source Files: Modular programming separates functions into.c files and.h files with function prototypes and declarations. Separation simplifies code management and lowers compiler dependencies.

Static and External Functions: The `static` keyword in C restricts a function's scope to its defined file. External functions can be used in numerous files. The use of these keywords helps encapsulate and organize code.

```
// static function
static int internalFunction() {
    // code
}
// external function
extern int externalFunction();
```

Libraries and Code Reusability: Libraries contain precompiled routines that can be linked to other programs. Library creation and use improve code reuse, especially for project-wide functions.

CONCLUSION

C balances simplicity, efficiency, and control as a timeless maestro of programming languages. Its simplicity and low-level manipulation provide a solid framework for clearly expressing complicated ideas. Manage flow techniques, manage program execution, and provide accurate tools for dynamic applications. C's stack-heap memory management gives developers unmatched control over resources but requires careful planning to avoid hazards. C functions construct modular solutions and express a mindset that transcends individual lines of code. C programming emphasizes code reusability and maintainability through modular principles. C is a foundational guide to programming craftsmanship, not just a language. This attitude influences the art and science of software development across many areas and lines of code. Adroit programmers use C to mold the digital landscape, leaving an enduring stamp on technology.

REFERENCES

Bergel, A., Harrison, W., Cahill, V. Clarke, S. (2011). FlowTalk: Language Support for Long-Latency Operations in Embedded Devices. *IEEE Transactions on Software Engineering*, *37*(4), 526-543. https://doi.org/10.1109/TSE.2010.66.

Bošanský, M., Patzak, B. (2016). Different Approaches to Parallelization of Sparse Matrix Assembly Operation. *Applied Mechanics and Materials*, *825*, 91-98. https://doi.org/10.4028/www.scientific.net/AMM.825.91

Chrysafiadi, K., Virvou, M. (2013). Dynamically Personalized E-Training in Computer Programming and the Language C. *IEEE Transactions on Education*, *56*(4), 385-392. https://doi.org/10.1109/TE.2013.2243914

Hong, L. X., Li, C. M. (2013). Web Based E-Learning System for C-Language. *Applied Mechanics and Materials*, *427-429*, 2870. https://doi.org/10.4028/www.scientific.net/AMM.427-429.2870

Pei, Y. J., Mu, J. L., Zhang, L., Li, X. F., Jin, M. (2014). Simulation Realization of LV SVG Control Algorithm Based on EMTDC and C Language Mixed Programming. *Applied Mechanics and Materials*, *651-653*, 1093-1096. https://doi.org/10.4028/www.scientific.net/AMM.651-653.1093

Sui, Y., Ye, D., Xue, J. (2014). Detecting Memory Leaks Statically with Full-Sparse Value-Flow Analysis. *IEEE Transactions on Software Engineering*, *40*(2), 107-122. https://doi.org/10.1109/TSE.2014.2302311

Xiao, X. P. (2013). Bubbling Sort Algorithm Application Based on C Programming Language. *Applied Mechanics and Materials***,** *380-384*, 1734. https://doi.org/10.4028/www.scientific.net/AMM.380-384.1734

Yordzhev, K. (2013). The Bitwise Operations Related to a Fast Sorting Algorithm. *International Journal of Advanced Computer Science and Applications***,** *4*(9). https://doi.org/10.14569/IJACSA.2013.040917