ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)

Volume 10 Issue 12 December 2023

# Unleashing the Potential of C++: Using Optimization Techniques on Procedural-Oriented Programming for Enhanced Efficiency

[1] Md. Faruk Abdullah Al Sohan, [2] Kazi Redwan, [3] Mustakim Ahmed

[1] [2] [3] Department of Computer Science and Engineering American International University- Bangladesh (AIUB) Dhaka, Bangladesh
Corresponding Author Email: [1] farukabdullahh@gmail.com, [2] 22-46069-1@student.aiub.edu, [3] ahmedmustakim031@gmail.com

***Abstract*— This research explores the impact of optimization techniques on the performance of C++ code, focusing on the Fibonacci problem. We investigate the importance of efficient coding practices in achieving optimal solutions, particularly for large datasets. By utilizing techniques like memoization, loop unrolling, and function inlining, we aim to enhance the efficiency and effectiveness of our C++ code. Through performance comparisons between optimized and unoptimized solutions, we demonstrate the superiority of optimized code in terms of speed and resource utilization. Our findings underscore the significance of code optimization in obtaining efficient solutions for computational problems.**

***Index Terms*— c++ optimization techniques, unoptimized code, memoization, loop unrolling, and function in linin.**

## I. INTRODUCTION

Lung the mid-20th century, researchers and computer scientists initiated a systematic exploration and development of optimization techniques, delving into various methods and algorithms to enhance code optimization and program performance. Code optimization is basically a method or approach used to enhance the functionality, effectiveness or other desirable properties of a code [1]. In the context of C++ programming, optimization plays a critical role in unleashing the full potential of the language and achieving optimal code performance [2].

This research paper focuses on exploring C++ optimization techniques within the procedural-oriented programming (POP) paradigm. Our objective is to investigate the impact of optimization techniques on code performance and demonstrate their effectiveness in enhancing the efficiency of C++ programs. To illustrate the benefits of optimization, we consider the Fibonacci problem as a case study. The Fibonacci sequence, with its recursive nature and exponential growth, presents a computational challenge that necessitates optimized solutions for efficient execution. Throughout this research, we explore into various optimization techniques specifically tailored for C++ code. These techniques include memoization, function unrolling, and inlining, among others. By employing these techniques and conducting comprehensive experiments, we analyze and compare the performance of optimized code against unoptimized code, highlighting the advantages of optimization in terms of speed, resource utilization, and scalability [1,3].

The rest of the paper is organized as follows: In section II, the previous works regarding the C++ optimization techniques. In section III, we will show the details of the optimization paradigms with detailed figure and information. In section IV, optimization process and section V describes the working principle. Section VI will show the result and analyze the improvement these techniques provided and construct a recommended coding pattern. Section VII concludes the paper.

## II. LITERATURE REVIEW

A lot of research has been done related to the optimization of code; some was done in specific sectors such as mobile applications and embedded systems, and most of the research was based on the OOP concept.

A researcher published a paper in 2006 about C++ optimization for mobile applications. This researcher's work aimed to optimize object-oriented programming (OOP)-based code in mobile applications, with a particular focus on enhancing performance [4]. The researcher presented a range of optimization techniques and discussed their applicability in the context of mobile application development. The researcher's paper sheds light on various optimization techniques, providing insights into their usage and effectiveness. However, one aspect that appeared less elucidated was the analysis methodology employed by the researcher. The paper did not explicitly elaborate on how the results were derived or the specific approach used to arrive at the optimized code that exhibited superior performance. While most research in the field of C++ optimization has traditionally focused on object-oriented programming (OOP), the significance of exploring POP-based problem-solving approaches cannot be overlooked. [5]

ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)

Volume 10 Issue 12 December 2023

Considering all these issues, we have undertaken this work on C++ optimisation techniques within procedural-oriented programming (POP) to unlock the full potential of the language. Throughout our research, we have applied and thoroughly discussed various code optimization techniques, specifically tailored to enhance the performance and efficiency of POP-based C++ code. To validate the effectiveness of these optimizations, we have applied tools such as <chrono> for checking the execution time of code and also applied code profiling tools for in-depth analysis (iterations/ejections/bytes). Finally, we have expressed that the optimized code we have developed works exceptionally well in handling large-scale data calculations.

## III. OPTIMIZATION CRITERIA

In our research, we delve into four key aspects of optimization: memory optimization, compiler optimization, algorithmic optimization, and I/O optimization. By addressing these areas, we aim to significantly improve the performance of our code. To analyze and evaluate the impact of these optimizations, we rely on two crucial tools: <chrono> for measuring execution time and gperftools for CPU profiling [6,7].

To conduct CPU profiling using gperftools, we utilize the following commands in the terminal of macOS:

1. Compilation:
   g++ *filename.cpp* -o myfile -lprofiler
2. Profiling command:
   CPUPROFILE=myfile.prof./myfile [7]

The profiles obtained provide detailed insights into the code's performance, enabling us to identify areas for improvement.

To analyze the performance of our code, we employ <chrono> to measure the execution time of specific code snippets. By strategically placing <chrono> timers before and after the code under examination, we can accurately measure the time taken for execution. We repeat the execution multiple times to ensure consistency and reliable results. [6]

Here's a pseudocode example demonstrating the usage of <chrono> for performance analysis:

```
#include <iostream>
#include <chrono>
using namespace std;
int main() {
   chrono::high_resolution_clock::time_point startTime, endTime;
   startTime = chrono::high_resolution_clock::now();
            // Code snippet to be tested
         // ...
   endTime = chrono::high_resolution_clock::now();
   chrono::duration<double, milli> duration = endTime – startTime;
      return 0;
} [6]
```

### A. Memory Optimization

Memory optimization is important in procedural-oriented programming (POP), especially when working with limited resources. When we don't use variables and memory efficiently, it can slow down our programs and make them less efficient. To solve this, we can use techniques that help us make the best use of the memory we have [1].

*Example:*

```
unsigned long long fibonacci (int n) {
   if (n <= 1) {
      return n;
   }
   unsigned long long prev = 0;
   unsigned long long curr = 1;

   for (int I = 2; I <= n; i++) {
      unsigned long long temp = curr;
      curr = prev + curr;
      prev = temp;
   }
return curr;
}
```

#### *1) Minimizing memory usage:*

Minimizing memory usage is a memory optimization technique aimed at reducing the amount of memory required by a program. By minimizing memory usage, we can conserve valuable resources and improve the overall performance and efficiency of our code.

In this example, we have used the unsigned long long data type to accommodate larger Fibonacci numbers. We only store the previous and current Fibonacci numbers (*'prev'* and *'curr'*), minimizing the memory usage. [1]

#### *2) Efficient data structure utilization*

Efficient data structure utilization is a memory optimization technique that focuses on choosing and utilizing data structures in a way that maximizes memory efficiency and performance. By selecting appropriate data structures and using them efficiently, we can minimize memory usage and improve the overall efficiency of our code. [2]

In this example, we are using minimal memory by only storing the necessary variables *'prev'* and *'curr'*. There is no additional memory allocation or complex data structure utilization, resulting in efficient memory usage.

#### *3) Optimizing memory deallocation*

Optimizing memory deallocation is a memory optimization technique that focuses on efficiently managing and releasing memory resources when they are no longer needed. Proper memory deallocation helps prevent memory leaks and ensures that memory is freed up for other parts of the program to use. [2]

ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)

Volume 10 Issue 12 December 2023

In this example, we have used the *unsigned long long* data type to handle larger Fibonacci numbers. Since there is no dynamic memory allocation involved, explicit memory deallocation is not needed. The variables *'prev'* and *'curr'* will be automatically deallocated when they go out of scope.

*4) Use appropriate variable types:*

Choose the smallest variable type that can accommodate the required range of values. For example, use *'int'* instead of *'long'* when dealing with smaller numbers. This helps save memory by avoiding excessive storage space. [3]

## B. Compiler Optimization

Compiler optimization is the process of automatically improving the performance and efficiency of code during the compilation phase. It involves analyzing the code and applying transformations to generate optimized machine instructions that can execute faster and use system resources more efficiently. [3]

*1) Loop Unrolling:*

Loop unrolling is a technique where multiple iterations of a loop are combined into a single iteration, reducing the overhead of loop control. In the case of the Fibonacci code, loop unrolling can be applied to compute multiple Fibonacci numbers in a single iteration.

Basically, we write loop in this way:

```
for (int I = 2; I <= n; i++) {
    int temp = curr;
    curr = prev + curr;
    prev = temp;
}
```

After unrolling the loop of our code it will be:

```
for (int I = 2; I <= n; I += 2) {
    int temp1 = curr + prev;
    int temp2 = curr + temp1;

    prev = temp1;
    curr = temp2;
}
```

In the optimized version with loop unrolling, instead of computing Fibonacci numbers one by one, we calculate two Fibonacci numbers in each iteration. This reduces the number of loop iterations by half, resulting in improved performance. In the benchmark test with n=10000, the original Fibonacci loop took approximately 20 microseconds to execute, while the unrolled loop only took 1 or 2 microseconds. This demonstrates the significant performance improvement achieved by using loop unrolling technique, reducing the execution time by a factor of 10 [1,2].

*2) Function Inlining:*

Function inlining is a compiler optimization technique where the code of a called function is directly inserted into the calling function, eliminating the overhead of function calls. In the case of the Fibonacci code, inlining the recursive function call can improve performance.

Example of Unoptimized Version:

```
int fibonacci_recursive_unoptimized(int n) {
    if (n <= 1) {
        return n;
    }
    return fibonacci_recursive_unoptimized(n – 1) + fibonacci_recursive_unoptimized(n – 2);
}
```

Example of Optimized Version with Function Inlining:

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

    int fibNMinus2 = 0;
    int fibNMinus1 = 1;

    for (int I = 2; I <= n; i++) {
        int fibN = fibNMinus1 + fibNMinus2;
        fibNMinus2 = fibNMinus1;
        fibNMinus1 = fibN;
    }
    return fibNMinus1;
}
```

In the optimized version with function inlining, the recursive function call is replaced with an iterative loop that directly calculates the Fibonacci number. This eliminates the overhead of function calls and improves performance.

In the benchmark test with n=10000, the original recursive Fibonacci implementation failed to provide an output, indicating a limitation in handling large values of n. However, the inlined Fibonacci calculation with a loop performed successfully and took approximately 26 microseconds to execute. This demonstrates the efficiency and improved performance of the inlined approach compared to the recursive implementation. [1,2]

*3) Constant Folding:*

Constant folding is a compiler optimization technique where expressions involving constants are evaluated at compile-time rather than runtime. In the case of the Fibonacci code, constant folding can be applied to optimize the calculation of the base cases. [9]

Example of Unoptimized Version:

```
int fibonacci_unoptimized(int n) {
    if (n <= 1) {
        return n;
    }
    return fibonacci_unoptimized(n – 1) + fibonacci_unoptimized(n – 2);
}
```

Example of Optimized Version with Constant Folding:

ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)

Volume 10 Issue 12 December 2023

```
int fibonacci_optimized_constant_folding(int n) {
    if (n <= 1) {
        return n;
    }
    const int sqrtFive = sqrt(5);
    const double phi = (1 + sqrtFive) / 2;

    return static_cast<int>((pow(phi, n) – pow(1 – phi, n)) /
sqrtFive);
}
```

In the optimized version with constant folding, the Fibonacci formula using the golden ratio is used to directly compute the Fibonacci number without the need for recursion or iterative loops. This results in improved performance. In the benchmark test with n=10000, the recursive Fibonacci algorithm failed to produce an output within a reasonable time frame, indicating its inefficiency for large inputs. On the other hand, the optimized version using constant folding completed the computation in approximately 15 to 17 microseconds. This demonstrates the significant performance improvement achieved by leveraging constant folding, reducing the execution time compared to the unoptimized recursive approach.

### 4) *Loop Fusion:*

Loop fusion is a technique where multiple loops are combined into a single loop, reducing loop overhead and improving cache utilization. In the case of the Fibonacci code, loop fusion can be applied to combine the calculation of Fibonacci numbers and their sum [8].

Example of Unoptimized Version:

```
int fibonacci_sum_unoptimized(int n) {
    if (n <= 1) {
        return n;
    }
    int sum = 0;
    for (int I = 0; I <= n; i++) {
        sum += fibonacci_unoptimized(i);
    }
    return sum;
}
```

Example of Optimized Version with Loop Fusion:

```
int fibonacci_sum_optimized_loop_fusion(int n) {
    if (n <= 1) {
        return n;
    }
    int sum = 0;
    int fibNMinus2 = 0;
    int fibNMinus1 = 1;

    for (int I = 2; I <= n; i++) {
        int fibN = fibNMinus1 + fibNMinus2;
        sum += fibN;
        fibNMinus2 = fibNMinus1;
        fibNMinus1 = fibN;
    }
    return sum + 1;
}
```

*In* the optimized version with loop fusion, the calculation of Fibonacci numbers and their summation is combined into a single loop. This reduces the number of iterations and eliminates the need for repetitive function calls, resulting in improved performance. In the benchmark test with n = 10000, the unoptimized Fibonacci sum function encountered issues and was unable to produce an output. However, the optimized Fibonacci sum function with loop fusion took approximately 22-27 microseconds to execute. This showcases a significant performance improvement compared to the unoptimized version, demonstrating the effectiveness of loop fusion in reducing the execution time for Fibonacci sum calculations. [2]

### 5) *Dead Code Elimination:*

Dead code elimination is a compiler optimization technique where unused or unreachable code is removed from the program. In the case of the Fibonacci code, dead code elimination can be applied to remove unnecessary calculations [8].

Example of Unoptimized Version:

```
int fibonacci_unoptimized(int n) {
    if (n <= 1) {
        return n;
    }
    int fib1 = 0;
    int fib2 = 1;
    int fibN = 0;
    for (int I = 2; I <= n; i++) {
        fibN = fib1 + fib2;
        fib1 = fib2;
        fib2 = fibN;
    }
    return fibN;
}
```

Example of Optimized Version with Dead Code Elimination:

```
int fibonacci_optimized_dead_code_elimination(int n) {
    if (n <= 1) {
        return n;
    }
    int fib1 = 0;
    int fib2 = 1;
    for (int I = 2; I <= n; i++) {
        int fibN = fib1 + fib2;
        fib1 = fib2;
        fib2 = fibN;
    }
    return fib2;
}
```

In the optimized version with dead code elimination, the unnecessary variable fibN is eliminated as it is not needed to compute the final Fibonacci number. This reduces memory usage and improves performance. In the benchmark test with n=100000, the unoptimized Fibonacci function took approximately 283 microseconds to execute, while the optimized version with dead code elimination only took 255 microseconds. This demonstrates the significant performance improvement achieved by eliminating dead code, reducing the execution time by approximately 10%. The results indicate that the optimized version performs better and is more efficient in calculating the Fibonacci sequence for larger values of n. [3]

*6) Common Subexpression Elimination:*

Common subexpression elimination is a compiler optimization technique where redundant computations are identified and eliminated. In the case of the Fibonacci code, common subexpressions can be identified and calculated only once.

Example of Unoptimized Version:

```
int fibonacci_unoptimized(int n) {
    if (n <= 1) {
        return n;
    }
    return fibonacci_unoptimized(n – 1) + fibonacci_unoptimized(n – 2);
}
```

Example of Optimized Version with Common Subexpression Elimination:

```
int fibonacci_optimized_common_subexpression_elimination(int n) {
    if (n <= 1) {
        return n;
    }
    int fibNMinus2 = 0;
    int fibNMinus1 = 1;

    for (int I = 2; I <= n; i++) {
        int fibN = fibNMinus1 + fibNMinus2;
        fibNMinus2 = fibNMinus1;
        fibNMinus1 = fibN;
    }
    return fibNMinus1;
}
```

In the optimized version with common subexpression elimination, the redundant computation of fibonacci_unoptimized(n–1) and fibonacci_unoptimized (n – 2) is eliminated. Instead, the Fibonacci numbers are calculated iteratively using two variables, fibNMinus2 and fibNMinus1. This reduces redundant function calls and improves performance. In the benchmark test with n = 10000, the original Fibonacci recursion took a significant amount of time and didn't produce an output. However, the optimized Fibonacci function with common subexpression elimination completed in approximately 26 microseconds. This demonstrates the effectiveness of the optimization technique in reducing the execution time and enabling the calculation of Fibonacci numbers for larger values of n.

### C. Algorithmic Optimization

Algorithmic optimization, also known as algorithmic efficiency, is a process that aims to improve the performance and efficiency of algorithms by minimizing unnecessary operations, reducing redundant computations, and utilizing available resources effectively. It is a crucial aspect of procedural-oriented programming (POP), which focuses on designing and implementing algorithms that are optimized in terms of time and space complexity. By analyzing the algorithm's structure, identifying bottlenecks, and making strategic modifications, algorithmic optimization plays a significant role in enhancing the efficiency and speed of software and systems.

Selecting or designing algorithms with lower time or space complexity is an important aspect of algorithmic optimization. In the context of the Fibonacci sequence, an optimized algorithm can be devised to achieve better performance.

Example of Unoptimized Version:

```
int fibonacci_unoptimized(int n) {
    if (n <= 1) {
        return n;
    }
    return fibonacci_unoptimized(n – 1) + fibonacci_unoptimized(n – 2);
}
```

Example of Optimized Version with Improved Time Complexity:

```
int fibonacci_optimized_time_complexity(int n) {
    if (n <= 1) {
        return n;
    }
    int fibNMinus2 = 0;
    int fibNMinus1 = 1;
    for (int I = 2; I <= n; i++) {
        int fibN = fibNMinus1 + fibNMinus2;
        fibNMinus2 = fibNMinus1;
        fibNMinus1 = fibN;
    }
    return fibNMinus1;
}
```

By eliminating the recursive calls and computing Fibonacci numbers iteratively, the optimized version achieves a lower time complexity compared to the unoptimized version. This results in improved performance when calculating Fibonacci numbers for large values of n. [2]

ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)

Volume 10 Issue 12 December 2023

*1) Reducing unnecessary computations:*

Unnecessary computations can impact the efficiency of an algorithm. In the case of the Fibonacci sequence, we can optimize the algorithm to avoid redundant calculations.

Example of Unoptimized Version:

```
int fibonacci_unoptimized(int n) {
    if (n <= 1) {
        return n;
    }
    return fibonacci_unoptimized(n – 1)
fibonacci_unoptimized(n – 2);
}
```

Example of Optimized Version with Reduced Computations:

```
int fibonacci_optimized_computations(int n) {
    if (n <= 1) {
        return n;
    }
    int fibNMinus2 = 0;
    int fibNMinus1 = 1;
    for (int I = 2; I <= n; i++) {
        int fibN = fibNMinus1 + fibNMinus2;
        fibNMinus2 = fibNMinus1;
        fibNMinus1 = fibN;
    }
    return fibNMinus1;
}
```

In the optimized version, the Fibonacci numbers are calculated only once and stored in variables (fibNMinus2 and fibNMinus1) to avoid redundant function calls. This reduces unnecessary computations and improves the efficiency of the algorithm. [1]

*2) Memoization to avoid redundant function calls:*

Memoization is a technique used to optimize recursive algorithms by storing the results of expensive function calls and reusing them when the same inputs occur again. It helps avoid redundant function calls and improves the overall efficiency of the algorithm. [1]

Example of Unoptimized Version:

```
int fibonacci_unoptimized(int n) {
    if (n <= 1) {
        return n;
    }
    return fibonacci_unoptimized(n – 1) +
fibonacci_unoptimized(n – 2);
}
```

Example of Optimized Version with Memoization:

```
unsigned long long fibonacci(int n, vector<unsigned long
long>& memo) {
    if (n <= 1)
        return n;

    if (memo[n] != -1)
        return memo[n];
        memo[n] = fibonacci(n – 1, memo) + fibonacci(n – 2,
memo);
    return memo[n];
}
```

In the optimized version, a memoization technique is applied using an unordered map (memo) to store previously computed Fibonacci numbers. This avoids redundant function calls for the same input values and improves the efficiency of the algorithm by reusing the stored results.

### D. I/O Optimization

I/O optimization in procedural-oriented programming (POP) languages like C++ aims to enhance the efficiency of input/output operations. Techniques such as buffering, sequential I/O, file access modes, and error handling are utilized to minimize overhead during external device or file operations. Specifically in C++, disabling the synchronization between C and C++ streams, using

***'ios_base::sync_with_stdio(false)'***

That improves I/O efficiency, particularly when exclusively using C++ input/output streams like cin and cout. This allows C++ streams to function independently, eliminating synchronization checks and potentially speeding up I/O operations. However, caution is advised to ensure exclusive use of C++ streams and avoid mixing them with C-style functions to prevent unexpected behavior [8].

## IV. PERFORMANCE ANALYSIS

In this research, our primary focus is on applying optimization techniques to the Fibonacci algorithm. By leveraging performance analysis tools like <chrono> and gperftools (CPU profiling), we aim to identify areas within the Fibonacci algorithm where performance improvements can be made. This may involve analyzing the time complexity of different approaches, examining memory usage, or profiling function calls to detect potential optimizations.

A. Performance Analysis Results using <chrono>: Unoptimized vs. Optimized Fibonacci Code:

| **n** | **Execution Time** (milliseconds) | |
|---|---|---|
| | ***Unoptimized Code*** | ***Optimized Code*** |
| 45 | 10000-12000 | 45 |
| 50 | 120000-130000 | 50 |
| 100 | >130000 | 100 |
| 500 | >130000 | 500 |
| 1000 | >130000 | 1000 |
| 5000 | >130000 | 5000 |

Sample of a Table footnote. (*Table footnote*)

ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)

Volume 10 Issue 12 December 2023

B. Performance Analysis Results Unoptimized Code using ‘gperftools (CPU Profiling)’:

| **n** | *CPU Profiling* | | |
|---|---|---|---|
| | ***Interrupts*** | ***Subhead*** | ***Bytes*** |
| 45 | 1200-1500 | 45 | >300000 |
| 50 | 120000-130 000 | 50 | >300000 |
| 100+ | >130000 | 100+ | >300000 |

C. Performance Analysis [ Code using ‘gperftools (CPU Profiling)’:

| **n** | *CPU Profiling* | | |
|---|---|---|---|
| | ***Interrupts*** | ***Subhead*** | ***Bytes*** |
| 0-900 0 | 0 | 0-9000 | 64 |
| 9000-20000 | 1 | 9000-20000 | 160 |

Optimization efforts are often directed towards improving program execution speed, reducing power consumption, optimizing bus bandwidth, and managing memory usage. However, these optimization goals are interconnected, and enhancing one aspect may inadvertently impact others. It’s important to acknowledge that even with the implementation of various optimization techniques, there is no guarantee of achieving overall program efficiency. Focusing efforts on optimizing operations within these functions can lead to noticeable performance improvements, ultimately benefiting the overall performance of the program.

## V. CONCLUSION

Optimizing code is essential for maximizing the performance and efficiency of C++ programs. By implementing memory optimization techniques, leveraging compiler optimizations, conducting thorough performance analysis, and carefully considering the trade-off between performance and maintainability, programmers can greatly enhance the execution speed and resource utilization of their code. While optimization does not guarantee universal efficiency, adopting these strategies facilitates the identification and implementation of improvements that contribute to better program performance. Embracing these recommendations is crucial for creating high-performing C++ applications that deliver optimal results, particularly when tackling complex computational problems and handling large datasets.

## REFERENCES

[1] EFFECTIVE C++ THIRD EDITION BY SCOTT MEYERS.

[2] C++ PRIMER 5TH EDITION BY STANLAY LIPPMAN.

[3] OPTIMIZED C++: PROVEN TECHNIQUES FOR HEIGHTENED PERFORMANCE 1ST EDITION BY KURT GUNTHEROTH.

[4] C++ optimization for mobile applications Fadi Chehimi, Paul Coulton and Reuben Edwards written on mobile application.

[5] “A review of using object-orientation properties of C++ for designing expert system in strategic planning Author links open overlay panel” Mohsen Ahmadi a, Moein Qaisari Hasan Abadi b.

[6] https://en.cppreference.com/w/cpp/chrono

[7] https://hackingcpp.com/cpp/tools/profilers.html

[8] TECHNIQUES FOR SCIENTIFIC C++ BY TODD VELDHUIZEN <TVELDHUI@ACM.ORG> VERSION 0.3, AUGUST 1999.

[9] https://iq.opengenus.org/constant-folding-and-propagation/https://iq.opengenus.org/constant-folding-and-propagation/