Codes in C++20 Included

# Programmers are Using Eager Evaluation on Daily Basis,

# While Professionals are Getting Advantage of Lazy Evaluation Technique

# Lazy Evaluation with Proxy Objects & Expression Templates; Fused Operations

# Included Benchmarks Show Lazy Evaluation Boosts Performance by 3900% 😮

*By: Pouya Ebadollahyvahed*

## 1- Questions are Answered in This Article

- What is "**Eager Evaluation**"?
  - All programmers are using Eager Evaluation in their daily life but may not heard the term
- What is "**Lazy Evaluation**" & How to Do It?
  - Using "**Proxy Objects**" to implement "Lazy Evaluation"
  - Built-in **STL** "**Fused Operations**"
- Developing an example **C++ code** to implementing "Lazy Evaluation" by "**Expression Templates**"
  - C++ Template **Meta-programming**
- What is **advantage** of "Lazy Evaluation" over "Eager Evaluation"?
- Which Famous Libraries in C++ are using "Lazy Evaluation" Beneath
- **Performance Benchmarking** of Eager & Lazy Evaluation in **Matrix Arithmetic** (**Neural Network** App)
  - Performance comparison in **Windows 11** and **Linux RHEL 9.5**
  - Performance comparison of **Visual Studio C++ MSVC**, **Intel DPC++**, **GNU g++** & **LLVM clang++**

## 2- What is **Eager Evaluation** & What is **Lazy Evaluation**?

"**Eager Evaluation**" or "**Greedy Evaluation**" means an operation (like a **function call** or an **operator call** such as `+`, `-`, `*`, `/`, `%`, `^`, `&`, `&&`, `|` and `||`) **should be performed as soon as it is encountered in the code**.

Eager Evaluation is normal & default way in programming. Here are two examples of Eager Evaluation:

- **Example 1:**

Consider following function call in `main(){ }` in the line before the last line. As soon as execution reaches this line, `f1()` will be called and then its returned value (=12) will be stored in a temporary variable and then `f2()` will be called and this temp variable will be passed to `f2()` function.

```
int f1(int i) {
   return i + 2;
}
int f2(int i, int j) {
   if (i < 50) return i + j;
   return i * 2;
}

int main() {
   cout << "Final Result is: " << f2( 55, f1(10) );
}
```

- **Example 2:**

Suppose `a`, `b`, `c` and `d` are variables (of type **int** or any other type) or objects (of any class overloading **operator+** and **operator=** in regular way) then expression

```
a = b + c + d;
```

will be evaluated in eager way that means when the execution reaches to this line, it does these steps:

1- `b` and `c` will be added (`b + c`) and the result will be stored in a temporary variable (like `s`)

2- `s` and `d` will be added (`s + d`)

3- The result of above will be stored in `a` (or in another temp variable and then moves to `a`)

Eager evaluation in above examples has 2 **drawbacks** & both of them are about **efficiency** & performance:

1- In example 1 of the previous page, both `f1()` and `f2()` functions are being called but was it really necessary to call function `f1()`? By checking code of function `f2()` we find out that the 2nd argument of `f2()` (`int j = f1(10)`) is used only when its 1st argument (`int i = 55`) is less than 50 (inside the `if` block). In this example, value of the 1st argument is 55 so the 2nd argument is not even used in the code, **so calling function `f1()` was not required**.

2- In example 2 of the previous page, **operator+** is called 2 times (because expression has 3 operands). If expression was like $a = b_0 + b_1 + \dots + b_N;$ then **operator+** would be called **N** times. In case operands ($b_i$) be complicated objects like matrices, each time of calling this (or any other) operator causes a great overhead and wastes a lot of processors' (CPU / GPU) time, while if we could be able to do all of the "summation operations" in 1 step (like summing up elements of all matrices in 1 `for` loop) it could save a lot of processors' (CPU / GPU) time & result in higher performance.

"**Lazy Evaluation**" **is a programming technique to increase execution performance.**

"**Lazy Evaluation**" is to **postpone performing of an operation** (like a **function call** or an **operator call** such as `+`, `-`, `*`, `/`, `%`, `^`, `&`, `&&`, `|` and `||`) **until its result is really needed**.

If we want to re-write codes of examples 1 & 2 of the previous page using Lazy Evaluation technique, then the codes should be changed in this way (full C++ codes of both examples are provided in next pages):

- **Example 1:** `f1()` should **not** be called when calling `f2()`. `f1()` should be called **ONLY** when its returned value (`int j = f1(10)`) is used (needed) inside body of `f2()`. Code is printed on page 5.
- **Example 2:** when execution reaches to any of **operator+**, it should **not** perform any action (since result is not needed). The **summation** operation should be performed when execution reaches to **operator=** and here all summations should be performed together (since only here result is needed).

## Re-writing Code of Example 1 in Two Versions: with Eager Evaluation & Lazy Evaluation Technique

Since example 1 is very simple, let's make it a bit more practical and then write the code in **C++20** in two versions:

1- Eager Evaluation

2- Lazy Evaluation

So, **now we want to develop a code for strings processing**. In our design, we will have 2 sets of functions:

- A set of low-level functions: Each function performs a basic "string operation" like finding a sub-string in the string.
- A set of high-level functions: Each function receives a string, checks it against specific rules to verify if it is a valid string (for example checks its length or language grammar or …) and if the string is being verified as valid, it calls the requested low-level function to perform string processing.

The above design is good **because to perform any string processing we just need to write 1 line of code**:

1- We call a high-level function and pass **a string** and **a low-level function** to it
    a. The high-level function verifies the string
    b. The low-level function processes the string
    c. If the string is being verified as "valid" the result of string processing will be printed
        i. If the string is **NOT** being verified as "valid", just an error message will be printed

The whole code is printed on the next page. We have 1 low-level function and 1 high-level function but the high-level function is implemented in 2 versions: with Eager Evaluation & Lazy Evaluation technique.

```cpp
// a simple example of Eager and Lazy function call

#include <iostream>
#include <functional>

using namespace std;


// A low-level function which does a fundamental job (finding subString "Test" in str)
int f_find_substring(std::string s){
	cout << "f_find_substring: I am called with argument: " << s << "\n";

	return s.find("Test");
}


// A high-level function which checks & processes a string by calling a low-level func
void f_eager_evaluation(string s, int i)
{
	if (s.size() < 10)
		cout << "f_eager_evaluation: too short sentence";
	else
	{
		int pos_found{ i };
		cout << "f_eager_evaluation: The word (Test) found at " << pos_found;
	}
	cout << "\n\n";
}


// A high-level function which checks & processes a string by calling a low-level func
void f_lazy_evaluation(string s, function<int(string)> f)
{
	if (s.size() < 10)
		cout << "f_lazy_evaluation: too short sentence";
	else
	{
		int pos_found{ f(s) };
		cout << "f_lazy_evaluation: The word (Test) found at " << pos_found;
	}
	cout << "\n\n";
}


// The main function
int main() {
	std::string s1{ "a Test in a longer sentence" }, s2{ "a Test" };

	std::function<int(std::string)> f_wrapper = f_find_substring;


	cout << "\n\n====== Eager Evaluation =============\n";
	f_eager_evaluation(s1, f_wrapper(s1));
	f_eager_evaluation(s2, f_wrapper(s2));


	cout << "\n\n====== Lazy Evaluation =============\n";
	f_lazy_evaluation(s1, f_wrapper);
	f_lazy_evaluation(s2, f_wrapper);
}
```

If you run code of the previous page, you will see the following result on your console screen:

```
====== Eager Evaluation =============
f_find_substring: I am called with argument: a Test in a longer sentence
f_eager_evaluation: The word (Test) found at 2

f_find_substring: I am called with argument: a Test
f_eager_evaluation: too short sentence



====== Lazy Evaluation =============
f_find_substring: I am called with argument: a Test in a longer sentence
f_lazy_evaluation: The word (Test) found at 2

f_lazy_evaluation: too short sentence
```

By checking the above result, we find out that:

- When the string is a valid string (length >= 10), both high-level functions (eager and lazy evaluated versions) perform in similar way and the low-level function is being called.
- When the string is an invalid string (length < 10), both high-level functions (eager and lazy evaluated versions) detect it and print the correct error message on the screen while in eager evaluation version still the low-level function is being called (while it was not required). Lazy evaluation version of the high-level function doesn't call the low-level function in this case.

**Difference Between These 2 Versions:** **In Eager Evaluation version**, we call low-level function directly as an argument being passed to high-level function so C++ first calls the low-level function and stores its "Returned Value" in a temporary variable and then calls high-level function and passes this temporary variable to it then high-level function decides to use this variable or no based on its internal logic.

**In Lazy Evaluation version**, instead of passing "Returned Value" of low-level function to high-level function, we pass the low-level function itself so inside the high-level function it is called **ONLY** when its "Returned Value" is needed. When its "Returned Value" is not needed (the string is invalid) it is not called.

## 3- Implementing **Lazy Evaluation** Using **Expression Templates**

**In this section, it is described how to implement Example 2 of page 2 with Lazy Evaluation technique.**

**Full source code of implementing "Example 2" (both eager & lazy versions) can be found in section 6.**

**Performance of above 2 versions is measured & compared together & results are printed in section 5.**

What is **Expression Templates**?

> **Expression Templates are a C++ template metaprogramming technique**. **They are used to build structures to represent operations** (like summing operation performed by "**+**") **in compile time**.
>
> Expression Templates enables programmers to bypass normal & default operations (like summing operation performed by **operator+**) & postpone operation till result is really needed.

**In section 6, an actual implementation of Expression Templates in C++ 20 is provided**. This code uses "Lazy Evaluation" technique to calculate Matrix Summation. In this code **operator+** & **operator=** are overloaded. Check the code & pay attention that a great part of the process is happening in **Compile time**.

Expression Templates are widely used in C++ libraries especially in **linear algebra libs** to increase efficiency of Vector, Matrix and Tensor calculations. Examples of libraries which are using Expression Templates[1]:

| | | |
|---|---|---|
| • Dlib | • Armadillo | • Blaze |
| • Blitz++ | • Boost uBLAS | • Eigen |
| • POOMA | • Stan Math Lib | • xtensor |

[1] https://en.wikipedia.org/wiki/Expression_templates

As it can be seen in the code printed in section 6, **Expression Templates postpone** summing-up **operations** (**operator+**) of matrices till **operator=** is being called and only in this stage all summing-up operations (**operator+**) are being performed together.

Performing several operations in 1 step (together) is also called "**Fused Operations**" by **Bjarne Stroustrup** in the latest version of his book, "The C++ Programming Language".

To perform several arithmetic operations together, there are also several built-in functions in C++ STL like **FMA** functions (**Fused Multiplication & Addition**).

These functions calculate & return the value of **x * y + z**.

Examples of such functions are:

```
#include <cmath>

float       fma ( float x      , float y      , float z );
double      fma ( double x     , double y     , double z );
long double fma ( long double x, long double y, long double z );
float       fmaf( float x      , float y      , float z );
long double fmal( long double x, long double y, long double z );
```

## 4- Implementing Lazy Evaluation Using Proxy Objects

What is **Proxy Objects**?

In object-oriented languages, proxy classes are used to implement a **software design pattern** which is called proxy pattern. Proxy objects are instances of proxy classes. Proxy objects are useful and can be used in several scenarios and one of these scenarios is implementing "Lazy Evaluation".

In this scenario, **a proxy object substitutes a real operation** (for example "greater than" operation performed by **operator>**) **and stores operation information** (what operation & which operands) **without performing the operation**. Later when the actual result of the operation is needed a function of this object can be called which actually performs the operation and returns the result.

A number of C++ libraries are using proxy objects beneath. Proxy objects are not visible to users; they are just used internally to increase efficiency & performance.

In the code of next page, we define a class named `MyPoint`. This class represents a **point** on the screen (with Mathematical Cartesian Coordinates). It provides a simple function to return **distance** of the point to the screen-origin (X = 0, Y = 0). This "distance function" can be used:

- To get the distance and then for example show it to user
- To compare distances of 2 points (to the origin) and check which point is closer (to the origin)

In the code of next page, distance function of `MyPoint` class is implemented in 2 different versions:

- `distanceEager`: This is "Eager Evaluation" version & always calculates & returns the correct distance
- `distanceLazy` : This is "Lazy Evaluation" version. It doesn't calculate the distance. It returns a proxy object of type `PointProxy`, then whenever the "value of distance" is required, this proxy object, calculates the distance and returns it but in **efficient way**. Let's first check the code and then talk about it.

```cpp
// a simple example of using proxy objects to implement Lazy Evaluation

#include <iostream>

using namespace std;


class PointProxy {
public:
     PointProxy(const double x, const double y) : dist_squared {x * x + y * y} { };

     double getDistSquared(void) const { return dist_squared; };

     operator double() const {cout << " [SQRT Proxy] "; return sqrt(dist_squared); }

     bool operator<(const double d) const { return dist_squared < d * d; }
     bool operator>(const double d) const { return dist_squared > d * d; }

     bool operator<(const PointProxy d) const{return dist_squared<d.getDistSquared();}
     bool operator>(const PointProxy d) const{return dist_squared>d.getDistSquared();}

protected:
     const double dist_squared{};
};



class MyPoint {
public:
      MyPoint(const double px, const double py) : x{ px }, y{ py } {};

      double     distanceEager(void){cout <<" [SQRT Point] "; return sqrt(x*x+y*y);};
      PointProxy distanceLazy (void){ return PointProxy{ x, y }; };

protected:
      double x{}, y{};
};



int main() {
      MyPoint p1{ 3.0, 4.0 }, p2{7.0, 6.0};

      cout<<"Distance from the Origin is (Eager): "<< p1.distanceEager() << "\n";
      cout<<"Distance from the Origin is (Lazy) : "<< p1.distanceLazy() << "\n\n\n";


      if (p1.distanceLazy() < 6.0)
            cout << "Distance of p1 to origin is less than six - Lazy" << "\n\n";

      if (p1.distanceEager() < 6.0)
            cout<< "Distance of p1 to origin is less than six - Eager" << "\n\n\n\n";


      if (p1.distanceLazy() < p2.distanceLazy())
            cout<< "p1 is closer to origin than p2 - Lazy" << "\n\n";

      if (p1.distanceEager() < p2.distanceEager())
            cout<< "p1 is closer to origin than p2 - Eager" << "\n\n";
}
```

If you run code of the previous page, you will see the following result on your console screen:

```
Distance from the Origin is (Eager):  [SQRT Point] 5
Distance from the Origin is (Lazy) :  [SQRT Proxy] 5


Distance of p1 to origin is less than six - Lazy

 [SQRT Point] Distance of p1 to origin is less than six - Eager


p1 is closer to origin than p2 - Lazy

 [SQRT Point]  [SQRT Point] p1 is closer to origin than p2 - Eager
```

As it can be seen in the above screen-shot:

- `distanceEager`: Upon it is called, always calculates the correct distance and returns the result. The distance is calculated with this formula: $d = \sqrt{(x-0)^2 + (y-0)^2}$. The **drawback of this formula** is that, **calculating square root is very slow job** and brings down the efficiency.
- `distanceLazy` : Upon it is called, always returns a proxy object of type `PointProxy`. This object calculates distance-squared in its constructor with formula: `dist_squared`= $d^2 = (x-0)^2 + (y-0)^2$ & stores it.
    - Whenever this object is being casted to a fundamental type (`double`), the object calculates square root of `dist_squared` = $d^2$ variable and returns the correct distance (for example when we need to show the distance value to user)
    - Whenever distances of 2 points need to be compares, no further calculations are being performed and just their `dist_squared` = $d^2$ variables are being compared. **It is more efficient** & increases the performance.
    - Whenever distance of a point needs to be compared with a number, the proxy object also doesn't calculate square root of `dist_squared` = $d^2$. **It does a better and more efficient job**. It takes the number (to be compared with the distance of the point), makes it squared (to the power of 2) and then compares it with `dist_squared` = $d^2$. **The reason is that calculating $d^2$ is faster than $\sqrt{d}$ .**

## 5- Benchmarking Performance of Eager Evaluation & Lazy Evaluation

All along this article we always mentioned the main advantage of “Lazy Evaluation” over “Eager Evaluation” is **efficiency** & execution **performance**, so now it is time to make an actual comparison.

To make an actual comparison, “Example 1” of page 2 is implemented in **C++ 20** in two different versions: “Lazy Evaluation” & “Eager Evaluation”. Full source code of this simple program is listed in the next section.

This benchmarking program compares execution performance of adding 10 matrices (summing up 10 matrices and storing the total value in another matrix) and measures the time spent on this adding (summing) operation in 2 scenarios: “Lazy Evaluation” & “Eager Evaluation”.

This program implements 3 matrix classes:

- `template<ArithmeticType data_type_of_elements, size_t ROWS, size_t COLS> class Matrix`
    - It is just a parent (base) class for following 2 classes & implements common defination. Arithmetic opeartors are not overloaded here, so no useful objects can be instantiated. All elements of newly created objects (matrices) are initialized with random values.

- `class MatrixEager: public Matrix`
    - It is a concrete class. Objects instantiated from this class support “`+`” and “`=`” operations in eager evaluation manner, so we can instansiate objects from this class and do summation.

- `class MatrixLazy : public Matrix`
    - A concrete class like the above class but performs addition (`+`) in lazy evaluation manner.

To do benchmarking, we create 10 matrices with size of 4000 * 3000 elements & set elements type to `double` and then measure how much time is spent on “**Adding Operation**” on these 10 matrices.

**Note 1:** Full source code of the benchmarking program is printed in section 6

**Note 2:** Hardware & software specification of the "Test Platform" can be found in section 7

**Note 3:** All compilations are done in "Release Mode"

The benchmarking program compiled with different compilers and executed under different operating systems. Following table lists measured **time spent on adding-up (summing-up) operation** in each case.

| OS | Compiling | | Run in | Measured Time (ms) | | |
|---|---|---|---|---|---|---|
| | Compiler | Optimization Switch | | Lazy Evaluation | Eager Evaluation | Difference (≈%) |
| Linux | GNU g++ | -O3 | CMD - bash | 38 | 1505 | 3900 |
| | | - | | 242 | 4267 | 1700 |
| | LLVM clang++ | -O3 | | 40 | 8413 | - |
| | | - | | 231 | 11934 | - |
| Windows | Visual Studio - MSVC | /Ot | PowerShell Run as Admin | 42 | 1653 | 3900 |
| | | /Ot | CMD – Normal Mode | 86 | 5651 | 6200 |
| | Intel DPC++ (icpx) | /O2 | PowerShell Run as Admin | 40 | 1412 | 3500 |
| | | /O2 | CMD – Normal Mode | 84 | 5078 | 6000 |

**Findings & Outcomes:**

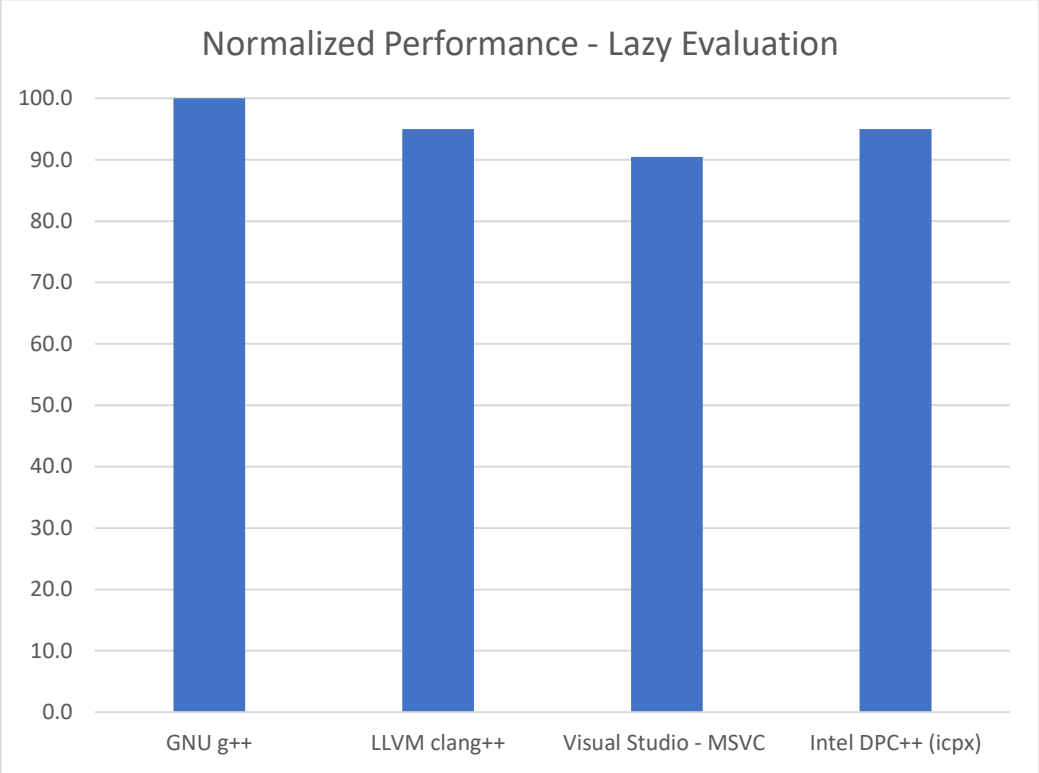
1- Performance of **Lazy Evaluation** versions is roughly **3900% better** than **Eager Evaluation** versions

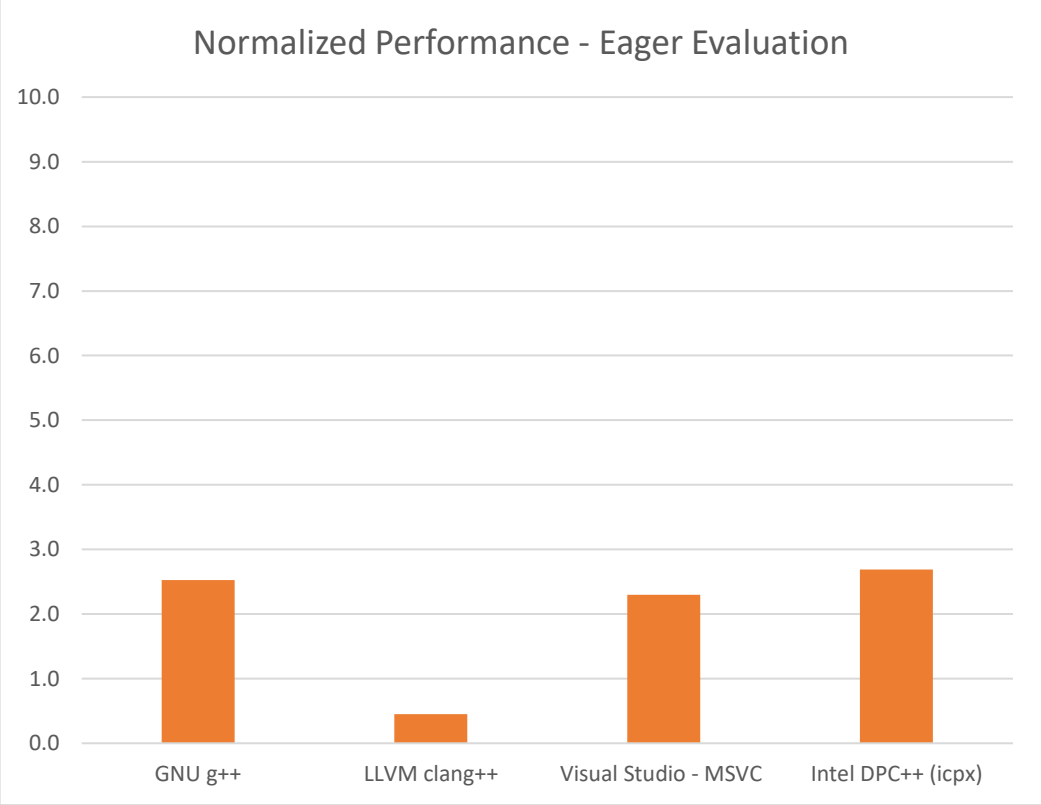
2- Performances of "**Lazy Evaluation**" version compiled with **different compilers** are **almost the same**

   a. Running the program in "command line - user mode" is 2 times slower than **admin-mode**

3- Performance of "**Not Optimized Compilation**" is **≈550%** lower than when using **-O3 optimization**

4- Performance of "**Eager Evaluation**" when the code is compiled with **LLVM is surprisingly Low (bad)**




**Note:** the formula for calculating "Normalized Performance" is $= 100 * \frac{\text{Shortest Measured Time (the Best)}}{\text{Time Measured in This Test}}$

## 6- Source Code of Performance Benchmarking Program (Using Expression Templates)

```
// This is companion code of the article for performance benchmarking of
//     "Eager Evaluation" & "Lazy Evaluation" in C++ 20
//     "Expression Templates" is used in this C++ code for "Lazy Evaluation"
//
//  This small project is ONLY for educational purposes & part of related article
//  Wherever in this project, member functions of classes were not needed for
//  the tests, they are not implemented
// File name: main.cpp (1 out of 4 project files)

#include "matrix_eager.h"
#include "matrix_lazy.h"
#include <chrono>

const size_t matrix_rows{ 4000 }; // Number of Rows    of all Matrices
const size_t matrix_cols{ 3000 }; // Number of Columns of all Matrices

/*****************************************************************************/
/*                              The Main Function                            */
/*****************************************************************************/


int main() {

      //This is an educational code, otherwise we must catch possible exceptions
      //       may be thrown in the following line if memory cannot be allocated

      // Uncomment only 1 of the following 2 lines to run the "performance test"
//    MatrixEager<double, matrix_rows, matrix_cols> m[10], m_total;
      MatrixLazy<double, matrix_rows, matrix_cols> m[10], m_total;

      cout << "\n\n== Matrix Operation (Summation) Started ==================\n\n";
      auto tStart = chrono::high_resolution_clock::now();

      m_total = m[0] + m[1] + m[2] + m[3] + m[4] + m[5] + m[6] + m[7] + m[8] + m[9];

      auto tEnd = chrono::high_resolution_clock::now();
      cout << "\n\n== Matrix Operation (Summation) Ended ====================\n\n";



      cout << "\n\n\n=================== Test Results ====================\n\n\n";

      const chrono::duration<double, milli> time_passed = tEnd - tStart;
      cout << "Time Spent on Operations: " << (size_t)time_passed.count() << " ms\n";


      const size_t row_no{ 0 }; // Row    number of the element to be printed
      const size_t col_no{ 0 }; // Column number of the element to be printed

      cout << "\nSample of summation of a matrix element:\n" << m[0][row_no][col_no];

      for (int mtrx_indx = 1; mtrx_indx < sizeof(m) / sizeof(m[0]); mtrx_indx++)
            cout << " + " << m[mtrx_indx][row_no][col_no];

      cout << " = " << m_total[row_no][col_no] << "\n\n";
}
```

```cpp
// This small C++ project is ONLY for educational purposes & part of related article
// File name: matrix.h (2 out of 4 project files)

#pragma once

#include <iostream>
#include <random>

/*****************************************************************************/
/*                          Definition - Globals                             */
/*****************************************************************************/

using namespace std;

template<typename T> concept ArithmeticType = is_arithmetic_v<T>;

// Just to generate some random numbers, 50 & 11 are also random
const float rnd_nums_mean{ 50.0f }, rnd_nums_std_dev{ 11.0f };


/*****************************************************************************/
/*                       Definition - Class - Matrix                         */
/*****************************************************************************/

// Although following is a concrete class but it is defined ONLY to be parent of
// "MatrixLazy" & "MatrixEager" classes & we instantiate only from these 2 classes

template<ArithmeticType T, size_t ROWS, size_t COLS>
class Matrix {
public:
	using value_type = T;

	Matrix() {
		pBuffer = new T[ROWS * COLS];

		normal_distribution<T>  rnd{ (T)rnd_nums_mean, (T)rnd_nums_std_dev };
		for (size_t pos = 0; pos != ROWS * COLS; ++pos)
			pBuffer[pos] = rnd(rnd_eng);
	}

	~Matrix() {
		delete[] pBuffer;
	}

	Matrix(const Matrix&) { cout << "Matrix Copy Constructor\n"; }
	Matrix(const Matrix&&) { cout << "Matrix Move Constructor\n"; }
	Matrix& operator=(const Matrix&) { cout << "Matrix copy =\n"; return *this; }
	Matrix& operator=(const Matrix&&) { cout << "Matrix move =\n"; return *this; }

	const T& get_element(const size_t pos) const { return pBuffer[pos]; }
	void set_element(const size_t pos, const T val) { this->pBuffer[pos] = val; }

	T* operator[] (const size_t row_num) { return (pBuffer + row_num * COLS); }

protected:
	T* pBuffer;

private:
	static inline default_random_engine rnd_eng{};
};
```

```
// This small C++ project is ONLY for educational purposes & part of related article
// File name: matrix_lazy.h (3 out of 4 project files)

#pragma once

#include "matrix.h"

/*****************************************************************************/
/*                     Definition - Class - SumProxy                         */
/*****************************************************************************/
template<typename LHS, typename RHS>
class SumProxy {
public:
	using value_type = typename RHS::value_type;

	SumProxy(const LHS& lh, const RHS& rh) : lhs{ lh }, rhs{ rh } {
//		cout << "SumProxy Default Constructor\n";
	}
	SumProxy(const SumProxy&) { cout << "SumProxy Copy Constructor\n"; }
	SumProxy(const SumProxy&&) { cout << "SumProxy Move Constructor\n"; }
	~SumProxy() {}

	SumProxy& operator=(const SumProxy&){cout<<"SumProxy Copy =\n"; return *this;}
	SumProxy& operator=(const SumProxy&&){cout<<"SumProxy Move =\n"; return *this;}

	value_type get_element(const size_t pos) const {
		return lhs.get_element(pos) + rhs.get_element(pos);
	}
private:
	const LHS& lhs;
	const RHS& rhs;
};

/*****************************************************************************/
/*                     Definition - Class - MatrixLazy                       */
/*****************************************************************************/
template<ArithmeticType T, size_t ROWS, size_t COLS>
class MatrixLazy : public Matrix<T, ROWS, COLS> {
public:
	template<typename LHS, typename RHS>
	MatrixLazy& operator=(const SumProxy<LHS, RHS>& rhs) {
//		cout << "\nMatrixLazy copy assignment with SumProxy argument\n\n";
		for (size_t pos = 0; pos != ROWS * COLS; ++pos)
			this->pBuffer[pos] = rhs.get_element(pos);

		return *this;
	}
};

/*****************************************************************************/
/*                     Overloading - Operators                               */
/*****************************************************************************/
template<typename LHS, typename RHS>
SumProxy<LHS, RHS> operator+(const LHS& lhs, const RHS& rhs) {
// An educational code, otherwise we must check equality of operands rows & columns
//	cout << "+ operator called, right operand: " << typeid(RHS).name() << "\n";
//	cout << "                   left operand: " << typeid(LHS).name() << "\n\n";
	return SumProxy<LHS, RHS>(lhs, rhs);
}
```

```cpp
// This small C++ project is ONLY for educational purposes & part of related article
// File name: matrix_eager.h (4 out of 4 project files)

#pragma once

#include "matrix.h"

/*****************************************************************************/
/*                    Definition - Class - MatrixEager                       */
/*****************************************************************************/


template<ArithmeticType T, size_t ROWS, size_t COLS>
class MatrixEager : public Matrix<T, ROWS, COLS> {

public:
	// This is an educational code, otherwise according to RULE-OF-FIVE, we had to
	// implement copy constructor, move constructor, move assignment & destructor

	MatrixEager& operator=(const MatrixEager& rhs) {

		for (size_t pos = 0; pos != ROWS * COLS; ++pos)
			this->pBuffer[pos] = rhs.get_element(pos);

		return *this;
	}


	MatrixEager operator+(const MatrixEager& rhs) {

		MatrixEager<T, ROWS, COLS> sum;

		for (size_t pos = 0; pos != ROWS * COLS; ++pos)
			sum.set_element(pos, this->get_element(pos)+rhs.get_element(pos));

		return sum;
	}
};
```

## 7- Test platform for Benchmarking (Hardware & Software Specification)

| | |
|---|---|
| **Machine** | Asus TUF Gaming F15 |
| **CPU** | Intel Core i7 12700 (6 P-Core + 8 E-Core, in total 20 cores) @ 4 GHz |
| **RAM** | 32 GB @ 3.2 GHZ |
| **OS** | Windows 11 Enterprise 64-bit (24H2)<br>Linux RHEL V9.5 64-bit *(installed on Hyper-V VM)* |
| **Compiler** | *For Windows:*<br>Visual Studio 2022 V17.13 (MSVC V19.43)<br>Intel icpx (dpc++) 2025.0<br>*For Linux:*<br>GNU g++ V11.5<br>LLVM clang++ V18.1 |
| **Compile Mode** | 64-bit<br>Release Mode |
| **Language** | ISO C++20 |

## 8- Table of Acronyms and Abbreviations

| Acronym / Abbreviation | Stands For |
|---|---|
| Bash | Bourne Again Shell |
| CMD | Command Line |
| cpp | C Plus Plus |
| CPU | Central Processing Unit |
| DPC++ | Data Parallel Compiler (for C++) |
| E-Core | Efficient Core (CPU) |
| FMA | Fused Multiplication & Addition |
| GNU | Gnu's Not Unix |
| GPU | Graphics Processing Unit |
| HZ | Hertz |
| ISO | International Standard Organization |
| Lib | Library |
| LLVM | Low Level Virtual Machine (former) |
| ms | milli second |
| MSVC | Microsoft Visual C++ |
| OS | Operating System |
| P-Core | Performance Core (CPU) |
| RAM | Random Access Memory |
| RHEL | Red Hat Enterprise Linux |
| STL | Standard Template Library |
| V | Version |
| VS | Visual Studio |

## 9- Disclaimer

The performance benchmarks and results presented in this article are based on specific tests conducted using GNU g++, LLVM clang++, Microsoft MSVC and Intel DPC++ compilers. The tests were performed using the author's developed C++ codes and represent the performance under the given conditions and configurations (like compilation switches). The results may vary based on different hardware, software versions, compiler optimizations, and other factors.

The author does not claim the absolute superiority of one compiler / hardware over another, as each compiler / hardware may have different strengths and weaknesses depending on the use case. The information provided is intended for educational and informational purposes only and should not be considered as professional or investment advice.

Readers are encouraged to conduct their own tests and research to validate the findings and make informed decisions based on their specific requirements and environments.

# 10- References


1- Viktor Sehr, Björn Andrist: **C++ High Performance**. Packt Publishing, 2018.

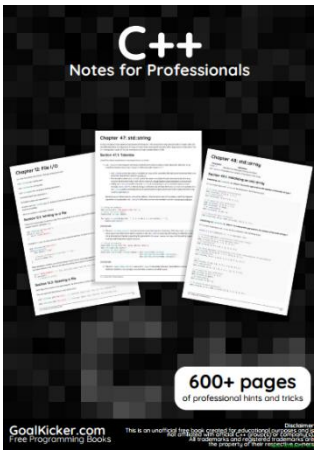
Chapter 9, Page 242




2- **C++ Notes for Professionals**. a Free eBook from

https://goalkicker.com/CPlusPlusBook.

Chapter 78, Page 443



3- https://gieseanw.wordpress.com/2019/10/20/we-dont-need-no-stinking-expression-templates/

4- https://en.wikipedia.org/wiki/Expression_templates