Take Home Test 3: Optimization

Shirong Zheng
Professor Izidor Gertner
April 5, 2017
CSC34200-G Spring 2017

Contents

❖1.Objective

The objective of take home test 3 is going to using the assembly code and C++ code which provide by Microsoft Visual Studio in 32 bit Window(debugger analysis) and GDB/GCC in 64 bit LINUX compiler. This topic of lab is factorial. That require to plot the time of it takes to compute Factorial(N), for N=10,100,1000,10,000. Using pointer arithmetic to access the array. Use index arithmetic to access elements in the array Section 2 Optimize with GCC in LINUX environment. Optimize product computation, (Instead of clear Array function) in Visual Studio environment.

❖2. SECTION 1 MS Visual Studio environment

The below is the general main file. That declare the array size as 10, the numbers are 1,2,3,4,5,6,7,8,9,-1

```
void ClearUsingIndex(int[], int);
void ClearUsingPointers(int*, int);

static int Array[10]={1,2,3,4,5,6,7,8,9,-1};

int main(){
    int size=10;
    ClearUsingIndex(Array,size);
    ClearUsingPointers(&array[0];size);
}
```

The below is my Index.cpp file. Let's set the array size equal to 0. The array, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

```
void ClearUsingIndex ( int Array[], int size){
    int i;
    for (i=0;i<size;i+=1)
        Array[i]=0;
}
```

The below is my pointers.cpp file. Let's set the pointer p of array equal to 0. a pointer is a programming language object, whose value refers to (or "points to") another value stored elsewhere in the computer memory using its memory address. A pointer references a location in memory, and obtaining the value stored at that location is known as dereferencing the pointer. As an analogy, a page number in a book's index could be considered a pointer to the corresponding page; dereferencing such a pointer would be done by flipping to the page with the given page number.

```
void ClearUsingPointers(int*array, int size){
    int *p;
    for (p=&array[0];p<&array[size];p=p+1)
        *p=0;
}
```

Next we going to generate two separate files into one assembly file. First we need to to call the Clear Using Index function. In the case, we could find out the Start Value and End Value, then use the math function " (End Value-Start Value)*1.0/frequency". That is the increment time.

```
main.cpp    Index.cpp
#include "stdafx.h"
#include <tchar.h>
#include <windows.h>
#include <iostream>
using namespace std;
void ClearUsingIndex(int[], int);
const int n = 100000000;
static int arr[n];

int main()
{
    __int64 ctr1 = 0, ctr2 = 0, freq = 0;
    int acc = 0, i = 0;
    int j;
    for (j = 0; j < n; j++)
        arr[j] = j;

    if (QueryPerformanceCounter((LARGE_INTEGER *)&ctr1) != 0)
    {
        ClearUsingIndex(arr, n);
        QueryPerformanceCounter((LARGE_INTEGER *)&ctr2);

        cout << "Start Value: " << ctr1 << endl;
        cout << "End Value: " << ctr2 << endl;

        QueryPerformanceFrequency((LARGE_INTEGER *)&freq);
        cout << "QueryPerformanceCounter minimum resolution: 1/Seconds." << freq << endl;
        cout << n << " Increment time: seconds." << (ctr2 - ctr1) * 1.0 / freq << endl; // changed size to n


        cout << "End Value - Start Value = " << ctr2 - ctr1 << endl;

    }
    else
    {
```

```
    {
        DWORD dwError = GetLastError();
        cout << "Error value = {0}" << dwError << endl;
    }

    system("PAUSE");

    return 0;
}
```

The initial array size is 4. The data that needs to be stored is 'pushed' into the stack and data to be retrieved is 'popped' out from the stack. Stack is a LIFO data structure, i.e., the data stored first is retrieved last. The memory space reserved in the stack segment is used for implementing stack. The system will push the ebp and sub esp from 204, the address is 00000ccH. In line 4 of code, it moves the memory address of eax. The "dword ptr" part is called a size directive. This page explains them, but it wasn't possible to direct-link to the correct section. Basically, it means "the size of the target operand is 32 bits", so this will bitwise-AND the 32-bit value at the address computed by taking the contents of the ebp register and subtracting four with 0.

```
PUBLIC  ?ClearUsingIndex@@YAXQAHH@Z         ; ClearUsingIndex
EXTRN   __RTC_InitBase:PROC
EXTRN   __RTC_Shutdown:PROC
;   COMDAT rtc$TMZ
rtc$TMZ SEGMENT
;__RTC_Shutdown.rtc$TMZ DD FLAT:__RTC_Shutdown
rtc$TMZ ENDS
;   COMDAT rtc$IMZ
rtc$IMZ SEGMENT
;__RTC_InitBase.rtc$IMZ DD FLAT:__RTC_InitBase
rtc$IMZ ENDS
; Function compile flags: /Odtp /RTCsu /ZI
;   COMDAT ?ClearUsingIndex@@YAXQAHH@Z
_TEXT   SEGMENT
_i$ = -8                        ; size = 4
_Array$ = 8                     ; size = 4
_size$ = 12                     ; size = 4
?ClearUsingIndex@@YAXQAHH@Z PROC            ; ClearUsingIndex, COMDAT
; File \\mac\home\desktop\original\index.cpp
; Line 1
    push    ebp
    mov ebp, esp
    sub esp, 204                ; 000000ccH
    push    ebx
    push    esi
    push    edi
    lea edi, DWORD PTR [ebp-204]
    mov ecx, 51                 ; 00000033H
    mov eax, -858993460             ; ccccccccH
    rep stosd
; Line 3
    mov DWORD PTR _i$[ebp], 0
    jmp SHORT $LN4@ClearUsing
$LN2@ClearUsing:
    mov eax, DWORD PTR _i$[ebp]
    add eax, 1
    mov DWORD PTR _i$[ebp], eax
$LN4@ClearUsing:
    mov eax, DWORD PTR _i$[ebp]
    cmp eax, DWORD PTR _size$[ebp]
    jge SHORT $LN1@ClearUsing
; Line 4
    mov eax, DWORD PTR _i$[ebp]
    mov ecx, DWORD PTR _Array$[ebp]
    mov DWORD PTR [ecx+eax*4], 0
    jmp SHORT $LN2@ClearUsing
$LN1@ClearUsing:
; Line 5
```

The below is the system using the function ClearUsingPointers in the code. The initial Start time, End time and frequency will be 0. Do the same thing from last process which is find out the increment time. In the case, the initial size of n will be 100000000. The main function will has the exactly code as the Index.cpp file.

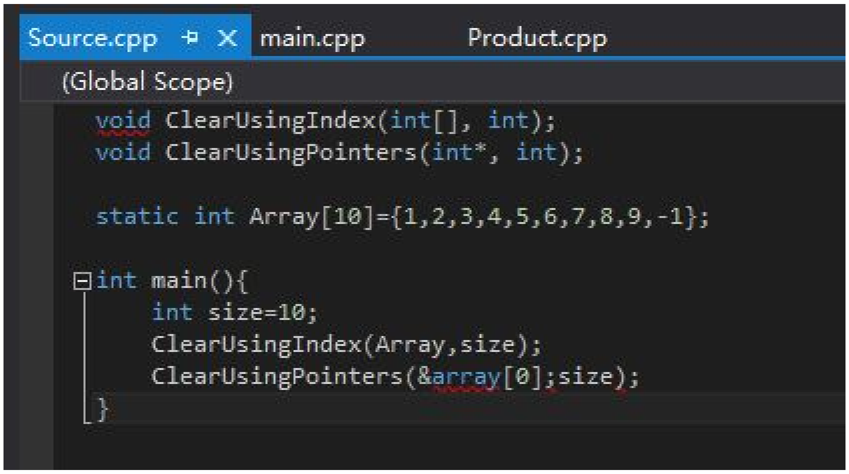
```
#include <tchar.h>
#include <windows.h>
#include <iostream>

void ClearUsingPointers(int*, int);

const int n = 100000000; //100000000;
static int arr[n];


using namespace std;

int main()
{
    __int64 ctr1 = 0, ctr2 = 0, freq = 0;
    int acc = 0, i = 0;
    int j;
    for (j = 0; j < n; j++)
        arr[j] = j;

    // Start timing the code.
    if (QueryPerformanceCounter((LARGE_INTEGER *)&ctr1) != 0)
    {
        // Code segment is being timed.

        int *p = arr;
        ClearUsingPointers(p, n);

        // Finish timing the code.
        QueryPerformanceCounter((LARGE_INTEGER *)&ctr2);

        cout << "Start Value: {0}" << ctr1 << endl;
```

```
        cout << "End Value: {0}" << ctr2 << endl;

        QueryPerformanceFrequency((LARGE_INTEGER *)&freq);

        //old version code: Console::WriteLine(S"QueryPerformanceCounter minimum resolution: 1/{0} Seconds.", freq.ToString());
        cout << "QueryPerformanceCounter minimum resolution: 1/{0} Seconds." << freq << endl;
        cout << n << " Increment time: {0} seconds." << (ctr2 - ctr1) * 1.0 / freq << endl; // changed size to n


        cout << "End Value - Start Value = " << ctr2 - ctr1 << endl;

    }
    else
    {
        DWORD dwError = GetLastError();
        //old version code:  Console::WriteLine(S"Error value = {0}", dwError.ToString());
        cout << "Error value = {0}" << dwError << endl;
    }

    // Make the console window wait.
    system("PAUSE");

    return 0;
}
```

Now let's move to the assembly part of the pointers. In line 3, the intended size of the of the data item at a given memory address can be inferred from the assembly code instruction in which it is referenced. For example, in all of the above instructions, the size of the memory regions could be inferred from the size of the register operand. When we were loading a 32-bit register, the assembler could infer that the region of memory we were referring to was 4 bytes wide. When we were storing the value of a one byte register to memory, the assembler could infer that we wanted the address to refer to a single byte in memory. However, in some cases the size of a referred-to memory region is ambiguous. Consider the instruction mov [ebx], 2. Should this instruction move the value 2 into the single byte at address EBX? Perhaps it should move the 32-bit integer representation of 2 into the 4-bytes starting at address EBX. Since either is a valid possible interpretation, the assembler must be explicitly directed as to which is correct. The size directives BYTE PTR, WORD PTR, and DWORD PTR serve this purpose, indicating sizes of 1, 2, and 4 bytes respectively.

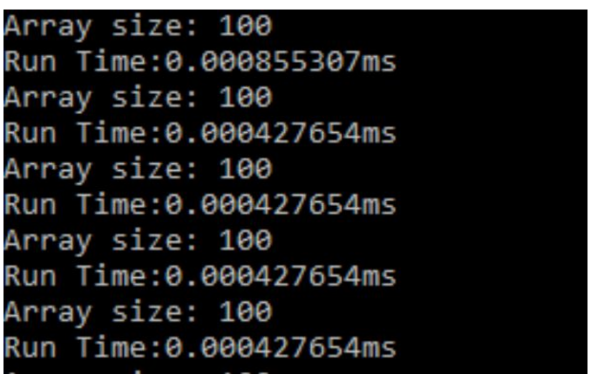
```
PUBLIC  ?ClearUsingPointers@@YAXPAHH@Z          ; ClearUsingPointers
EXTRN   __RTC_InitBase:PROC
EXTRN   __RTC_Shutdown:PROC
;   COMDAT rtc$TMZ
rtc$TMZ SEGMENT
;__RTC_Shutdown.rtc$TMZ DD FLAT:__RTC_Shutdown
rtc$TMZ ENDS
;   COMDAT rtc$IMZ
rtc$IMZ SEGMENT
;__RTC_InitBase.rtc$IMZ DD FLAT:__RTC_InitBase
rtc$IMZ ENDS
; Function compile flags: /Odtp /RTCsu /ZI
;   COMDAT ?ClearUsingPointers@@YAXPAHH@Z
_TEXT   SEGMENT
_p$ = -8                        ; size = 4
_array$ = 8                     ; size = 4
_size$ = 12                     ; size = 4
?ClearUsingPointers@@YAXPAHH@Z PROC         ; ClearUsingPointers, COMDAT
; File \\mac\home\desktop\original\pointers.cpp
; Line 1
    push    ebp
    mov ebp, esp
    sub esp, 204                ; 000000ccH
    push    ebx
    push    esi
    push    edi
    lea edi, DWORD PTR [ebp-204]
    mov ecx, 51                 ; 00000033H
    mov eax, -858993460             ; ccccccccH
    rep stosd
; Line 3
    mov eax, 4
    imul    ecx, eax, 0
    add ecx, DWORD PTR _array$[ebp]
```

The general main function, it declare the ClearUsingIndex and ClearUsingPointers.

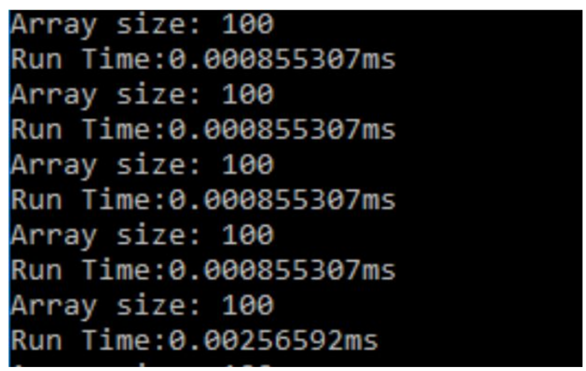



```
void ClearUsingIndex(int[], int);
void ClearUsingPointers(int*, int);

static int Array[10]={1,2,3,4,5,6,7,8,9,-1};

int main(){
    int size=10;
    ClearUsingIndex(Array,size);
    ClearUsingPointers(&array[0];size);
}
```

The first picture is original, the second is after optimization. When the array size is equal to 100, then it’s running time will be 0.000427654 in mostly time. After optimization, it will increase to 0.000855307. In the clusion, the the small array size optimization will cause lots time.



```
Array size: 100
Run Time:0.000855307ms
Array size: 100
Run Time:0.000427654ms
Array size: 100
Run Time:0.000427654ms
Array size: 100
Run Time:0.000427654ms
Array size: 100
Run Time:0.000427654ms
```



```
Array size: 100
Run Time:0.000855307ms
Array size: 100
Run Time:0.000855307ms
Array size: 100
Run Time:0.000855307ms
Array size: 100
Run Time:0.000855307ms
Array size: 100
Run Time:0.00256592ms
```

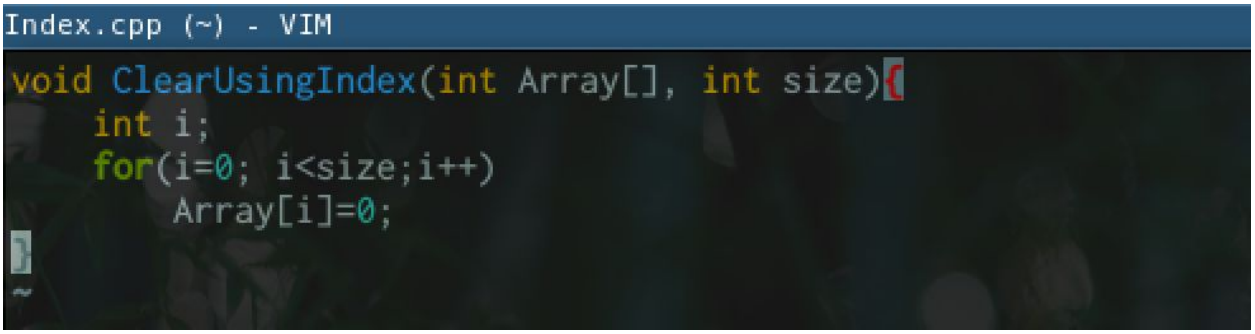
The first picture is original, the second is after optimization. When the array size is equal to 10000000, then it's running time will be around 28.3162 to 29.2844. After optimization, it will around 21.9695. In the clusion, the larger array size optimization will cause less time

```
Array size: 10000000
Run Time:28.3162ms
Array size: 10000000
Run Time:29.2844ms
Array size: 10000000
Run Time:29.2494ms
Array size: 10000000
Run Time:28.7961ms
Array size: 10000000
Run Time:28.3423ms
```

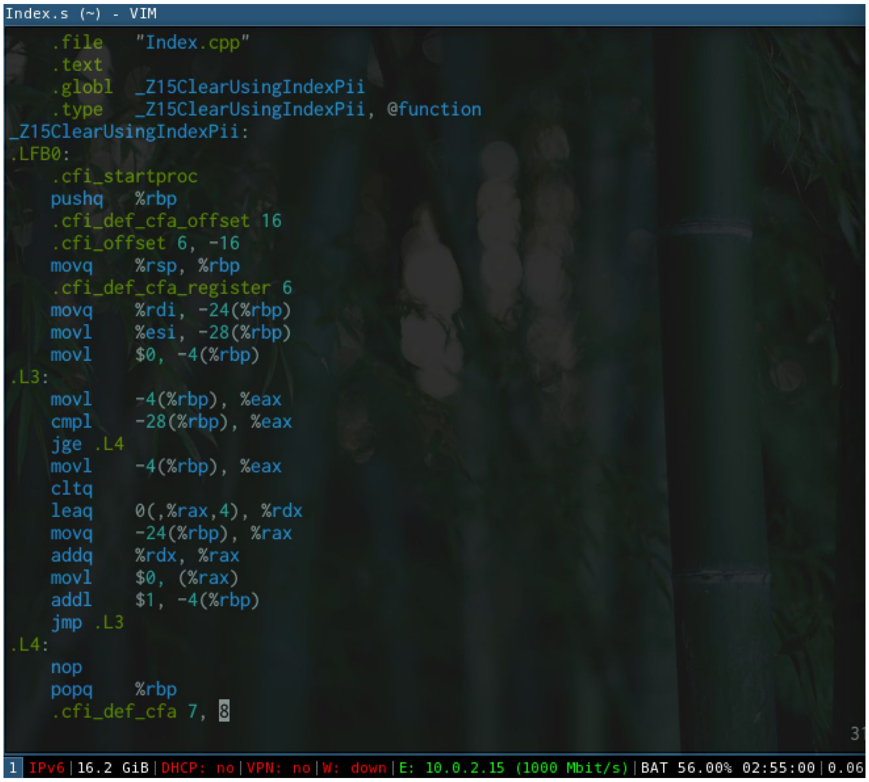
```
Array size: 10000000
Run Time:21.9536ms
Array size: 10000000
Run Time:21.9694ms
Array size: 10000000
Run Time:21.8796ms
Array size: 10000000
Run Time:21.9788ms
Array size: 10000000
Run Time:21.789ms
```

❖3. SECTION 2 GCC in LINUX environment

To finish this section, we need use gcc -S Index.cpp to execute the assembly code of Index.cpp file.

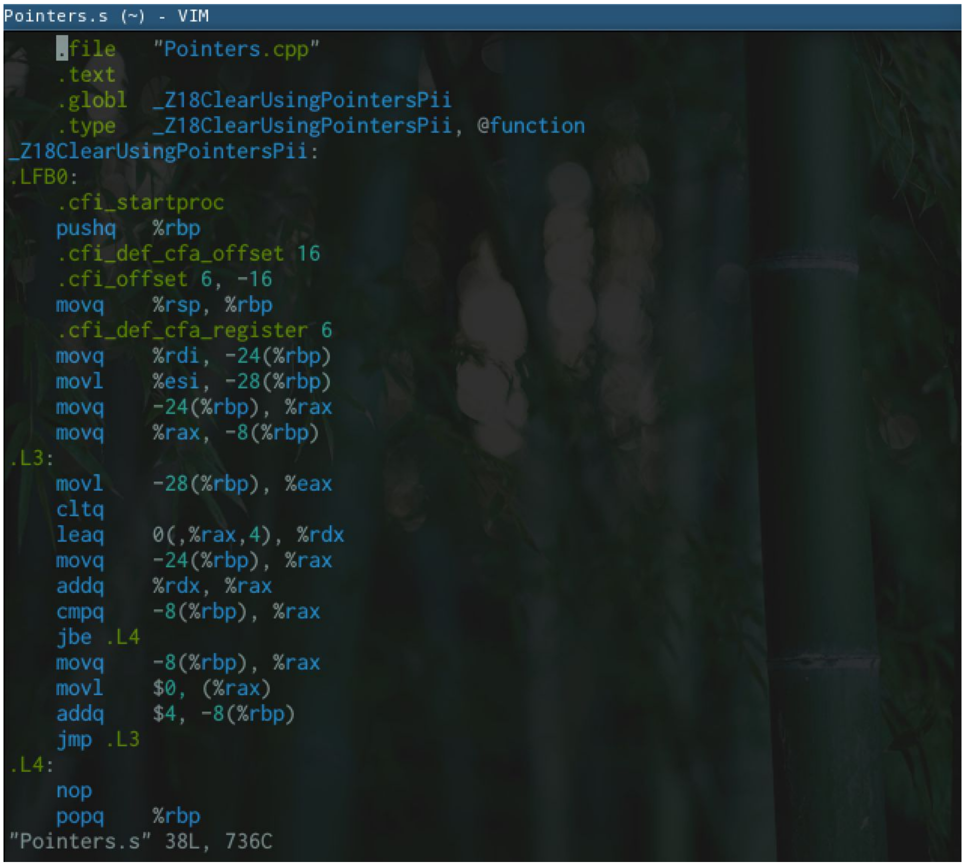


After compile, that shows the Index.s which is assembly code. The leftmost one is the source. For example, movl %edx, %eax means Move the contents of the edx register into the eax register. For another example, addl %edx,%eax means Add the contents of the edx and eax registers, and place the sum in the eax register.



Try to use gcc -S Pointers.cpp to execute the assembly code of Pointers.cpp file.

After compile, that shows the Index.s which is assembly code. cltq promotes an int to an int64. shl 3, %rax makes an offset to a 64-bit pointer (multiplies whatever is in rax by 8). what the code is doing is looping through a list of pointers to environment variables. when it finds a value of zero, that's the end, and it drops out of the loop.



```
Pointers.s (~) - VIM
	.file	"Pointers.cpp"
	.text
	.globl	_Z18ClearUsingPointersPii
	.type	_Z18ClearUsingPointersPii, @function
_Z18ClearUsingPointersPii:
.LFB0:
	.cfi_startproc
	pushq	%rbp
	.cfi_def_cfa_offset 16
	.cfi_offset 6, -16
	movq	%rsp, %rbp
	.cfi_def_cfa_register 6
	movq	%rdi, -24(%rbp)
	movl	%esi, -28(%rbp)
	movq	-24(%rbp), %rax
	movq	%rax, -8(%rbp)
.L3:
	movl	-28(%rbp), %eax
	cltq
	leaq	0(,%rax,4), %rdx
	movq	-24(%rbp), %rax
	addq	%rdx, %rax
	cmpq	-8(%rbp), %rax
	jbe .L4
	movq	-8(%rbp), %rax
	movl	$0, (%rax)
	addq	$4, -8(%rbp)
	jmp .L3
.L4:
	nop
	popq	%rbp
"Pointers.s" 38L, 736C
```

This project required to write the time.cpp file for the running time in the Linux. There is a math equation, time =(end.tv_sec -start.tv_sec)*1000000000ULL +end.tv_nsec -start.tv_sec.

```
time.cpp (~) - VIM
#include<stdio.h>
#include<stdint.h>
#include<stdlib.h>
#include<time.h>
using namespace std;
#define SIZE 100000000

void ClearUsingIndex(int[],int);
void ClearUsingPointers(int*,int);

static int Array[SIZE];

main(int argc, char **argv){
    uint64_t time;
    double avg=0.0;
    struct timespec start,end;
    for (int i=0; i<100; i++){
        clock_gettime(CLOCK_MONOTONIC,&start);


        ClearUsingIndex(Array,SIZE);
        ClearUsingPointers(&Array[0],SIZE);

        //recorded end time
        clock_gettime(CLOCK_MONOTONIC, &end);

        time=(end.tv_sec - start.tv_sec)*1000000000ULL +end.tv_nsec - start.tv_sec;
        printf("time= %f ns\n", (double) time);
        avg += time;
    }
    avg/=100;
```

```
    printf("size =%i, avg time =%f ns \n", SIZE, avg);
    return 0;
}
```

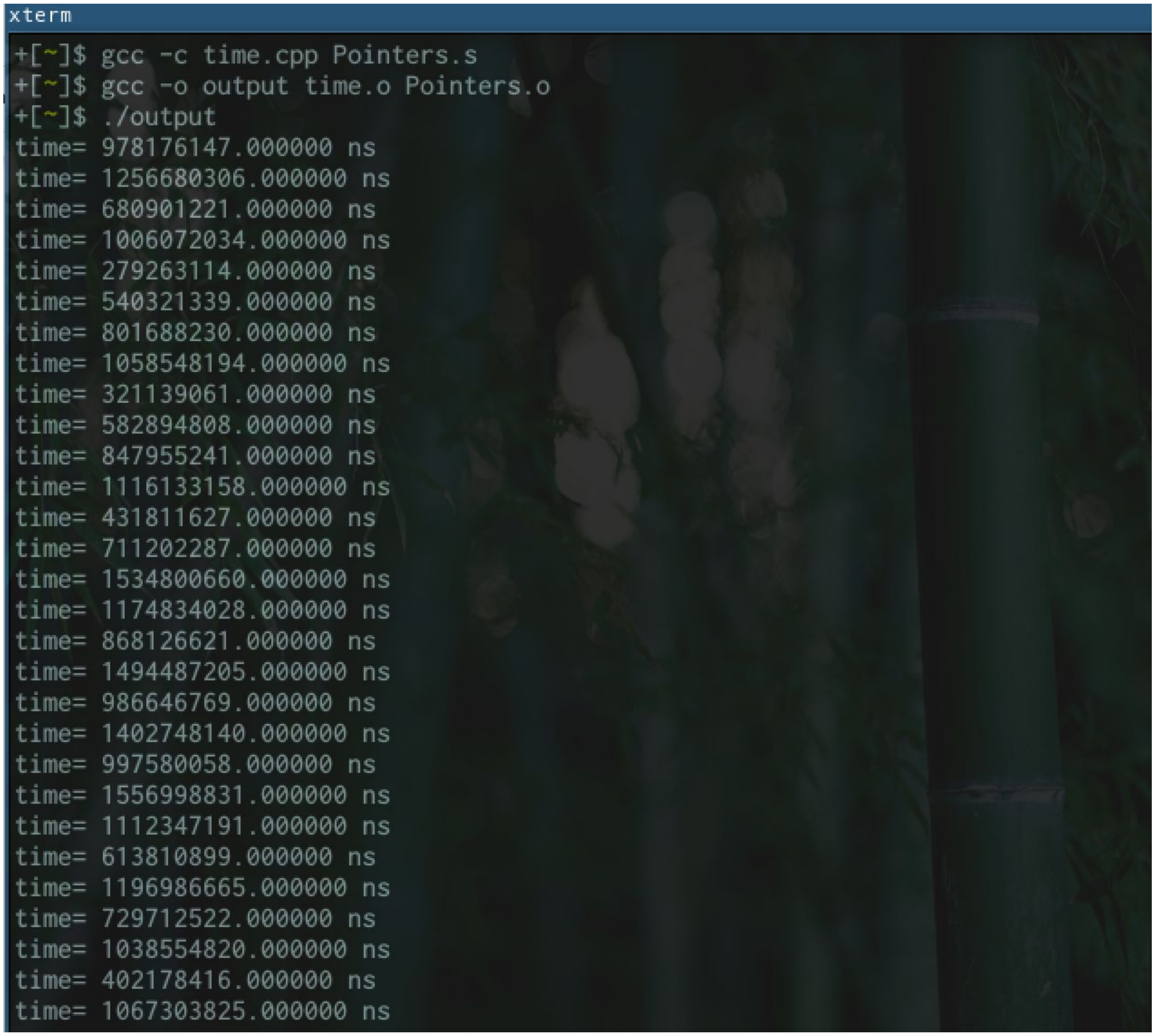
First type the command gcc -c timed.cpp function.s, this is to link the two source file together. Second type the command gcc -o output timed.o function.o, this is to link the object file and output a executable output file. Finally type command ./output.

The below is output of time.o and Index.o. The smallest running time is 1001876894 ns. The largest running time is 992755272 ns. The array size is 100000000, the average time is 80866826.9 ns.

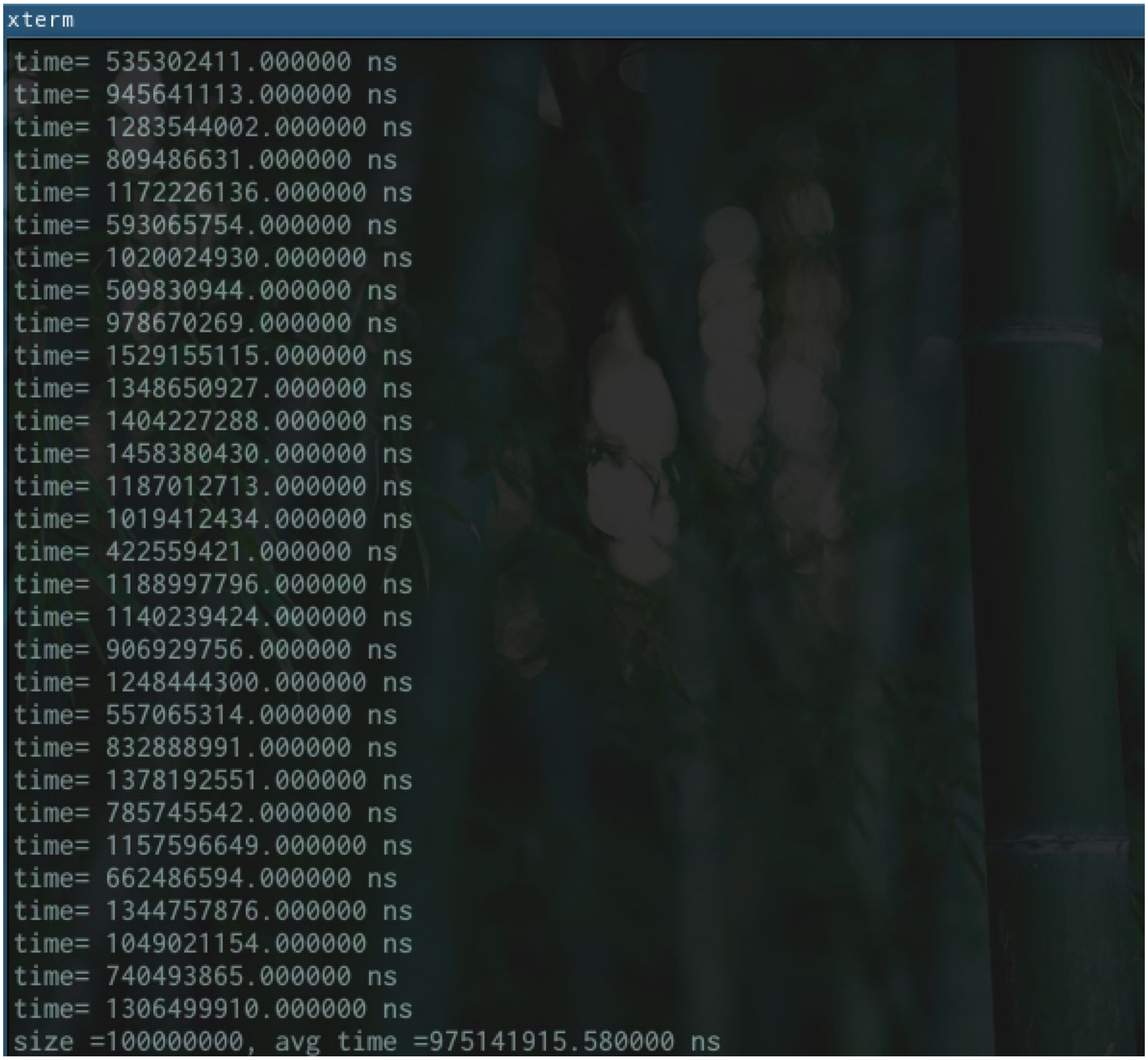
```
xterm
+[~]$ gcc -c time.cpp Index.s
+[~]$ gcc -o output time.o Index.o
+[~]$ ./output
time= 1253999313.000000 ns
time= 565596415.000000 ns
time= 885533891.000000 ns
time= 1192923309.000000 ns
time= 504564168.000000 ns
time= 876204818.000000 ns
time= 1186274399.000000 ns
time= 504074943.000000 ns
time= 814165767.000000 ns
time= 1120263033.000000 ns
time= 433714787.000000 ns
time= 750748146.000000 ns
time= 1068848790.000000 ns
time= 377219688.000000 ns
time= 688188362.000000 ns
time= 1001376894.000000 ns
time= 306503283.000000 ns
time= 608764759.000000 ns
time= 913864007.000000 ns
time= 1252650200.000000 ns
time= 560152529.000000 ns
time= 878309043.000000 ns
time= 1192464320.000000 ns
time= 510381298.000000 ns
time= 826495871.000000 ns
time= 1131274970.000000 ns
time= 467030737.000000 ns
time= 785039918.000000 ns
time= 1098110698.000000 ns
```

```
xterm
time= 1048537673.000000 ns
time= 354651797.000000 ns
time= 663667055.000000 ns
time= 972755272.000000 ns
time= 1274066460.000000 ns
time= 590534215.000000 ns
time= 891487863.000000 ns
time= 1203426543.000000 ns
time= 502093784.000000 ns
time= 811043203.000000 ns
time= 1137942908.000000 ns
time= 444911170.000000 ns
time= 757630298.000000 ns
time= 1085910542.000000 ns
time= 404927542.000000 ns
time= 714355332.000000 ns
time= 1015856178.000000 ns
time= 333969469.000000 ns
time= 646799223.000000 ns
time= 962135301.000000 ns
time= 1265340197.000000 ns
time= 581464736.000000 ns
time= 899645493.000000 ns
time= 1209806764.000000 ns
time= 509303704.000000 ns
time= 819761958.000000 ns
time= 1131984479.000000 ns
time= 437171400.000000 ns
time= 754481287.000000 ns
time= 1058077866.000000 ns
size =100000000, avg time =808668726.900000 ns
```

The below is output of time.o and Index.o. The below is output of time.o and Index.o. The smallest running time is 1006072034 ns. The largest running time is 9927580058 ns. The array size is 100000000, the average time is 975141915.58 ns.



```
xterm
+[~]$ gcc -c time.cpp Pointers.s
+[~]$ gcc -o output time.o Pointers.o
+[~]$ ./output
time= 978176147.000000 ns
time= 1256680306.000000 ns
time= 680901221.000000 ns
time= 1006072034.000000 ns
time= 279263114.000000 ns
time= 540321339.000000 ns
time= 801688230.000000 ns
time= 1058548194.000000 ns
time= 321139061.000000 ns
time= 582894808.000000 ns
time= 847955241.000000 ns
time= 1116133158.000000 ns
time= 431811627.000000 ns
time= 711202287.000000 ns
time= 1534800660.000000 ns
time= 1174834028.000000 ns
time= 868126621.000000 ns
time= 1494487205.000000 ns
time= 986646769.000000 ns
time= 1402748140.000000 ns
time= 997580058.000000 ns
time= 1556998831.000000 ns
time= 1112347191.000000 ns
time= 613810899.000000 ns
time= 1196986665.000000 ns
time= 729712522.000000 ns
time= 1038554820.000000 ns
time= 402178416.000000 ns
time= 1067303825.000000 ns
```



```
xterm
time= 535302411.000000 ns
time= 945641113.000000 ns
time= 1283544002.000000 ns
time= 809486631.000000 ns
time= 1172226136.000000 ns
time= 593065754.000000 ns
time= 1020024930.000000 ns
time= 509830944.000000 ns
time= 978670269.000000 ns
time= 1529155115.000000 ns
time= 1348650927.000000 ns
time= 1404227288.000000 ns
time= 1458380430.000000 ns
time= 1187012713.000000 ns
time= 1019412434.000000 ns
time= 422559421.000000 ns
time= 1188997796.000000 ns
time= 1140239424.000000 ns
time= 906929756.000000 ns
time= 1248444300.000000 ns
time= 557065314.000000 ns
time= 832888991.000000 ns
time= 1378192551.000000 ns
time= 785745542.000000 ns
time= 1157596649.000000 ns
time= 662486594.000000 ns
time= 1344757876.000000 ns
time= 1049021154.000000 ns
time= 740493865.000000 ns
time= 1306499910.000000 ns
size =100000000, avg time =975141915.580000 ns
```

❖4. SECTION 3 Performance measurement of Inner product computation

For INNER product computation *use expression*

$$(X, Y) = \sum_{i=0}^{N-1} x_i \, y_i$$

Where X, and Y are arrays of integers of size N.

The product is the x[i] time y[i] in the loop of n.

```
#include "stdafx.h"

int Product(int x[], int y[], const int n)
{
    int sum = 0;
    for (int i = 0; i < n; i++)
        sum += x[i] * (y[i]);
    return sum;
}
```

The below is the system using the function product in the code. The initial Start time, End time and frequency will be 0. Do the same thing from last process which is find out the increment time. In the case, the initial size of n will be 100000000. The main function will has the exactly code as before.

Source.cpp main.cpp Product.cpp

(Global Scope) main()

```cpp
#include <tchar.h>
#include <windows.h>
#include <iostream>

int Product(int[], int[], const int);

const int n = 100000000; //100000000;  100,000,000
static int x[n];
static int y[n];


using namespace std;

int main()
{
    __int64 ctr1 = 0, ctr2 = 0, freq = 0;
    int acc = 0, i = 0;
    int sum;
    int j;
    for (j = 0; j < n; j++){
        x[j] = 2;
        y[j] = 3;
    }
    // Start timing the code.
    if (QueryPerformanceCounter((LARGE_INTEGER *)&ctr1) != 0)
    {
        // Code segment is being timed.
        sum = Product(x, y, n);

        // Finish timing the code.
        QueryPerformanceCounter((LARGE_INTEGER *)&ctr2);
```

```cpp
        cout << "Start Value: {0}" << ctr1 << endl;
        cout << "End Value: {0}" << ctr2 << endl;

        QueryPerformanceFrequency((LARGE_INTEGER *)&freq);

        //old version code: Console::WriteLine(S"QueryPerformanceCounter minimum resolution: 1/{0} Seconds.", freq.ToString());
        cout << "QueryPerformanceCounter minimum resolution: 1/{0} Seconds." << freq << endl;
        cout << n << " Increment time: {0} seconds." << (ctr2 - ctr1) * 1.0 / freq << endl; // changed size to n


        cout << "End Value - Start Value = " << ctr2 - ctr1 << endl;

    }
    else
    {
        DWORD dwError = GetLastError();
        //old version code:  Console::WriteLine(S"Error value = {0}", dwError.ToString());
        cout << "Error value = {0}" << dwError << endl;
    }

    // Make the console window wait.
    system("PAUSE");

    return 0;
}
```

In this debug process, the system move value -858993460 to the eax, which has the memory address is ccccccccH. The "dword ptr" part is called a size directive. This page explains them, but it wasn't possible to direct-link to the correct section. Basically, it means "the size of the target operand is 32 bits", so this will bitwise-AND the 32-bit value at the address computed by taking the contents of the ebp register and subtracting four with 0. In the int sum=0, we could do the bitwise-AND operation.

```
      push    ebp
      mov ebp, esp
      sub esp, 216                ; 000000d8H
      push    ebx
      push    esi
      push    edi
      lea edi, DWORD PTR [ebp-216]
      mov ecx, 54                 ; 00000036H
      mov eax, -858993460             ; ccccccccH
      rep stosd

; 3   :   int sum = 0;

      mov DWORD PTR _sum$[ebp], 0

; 4   :   for (int i = 0; i < n; i++)

      mov DWORD PTR _i$1[ebp], 0
```

```
        mov eax, DWORD PTR _i$1[ebp]
        mov ecx, DWORD PTR _x$[ebp]
        mov edx, DWORD PTR _i$1[ebp]
        mov esi, DWORD PTR _y$[ebp]
        mov eax, DWORD PTR [ecx+eax*4]
        imul    eax, DWORD PTR [esi+edx*4]
        add eax, DWORD PTR _sum$[ebp]
        mov DWORD PTR _sum$[ebp], eax
        jmp SHORT $LN2@dot
    $LN1@dot:
```

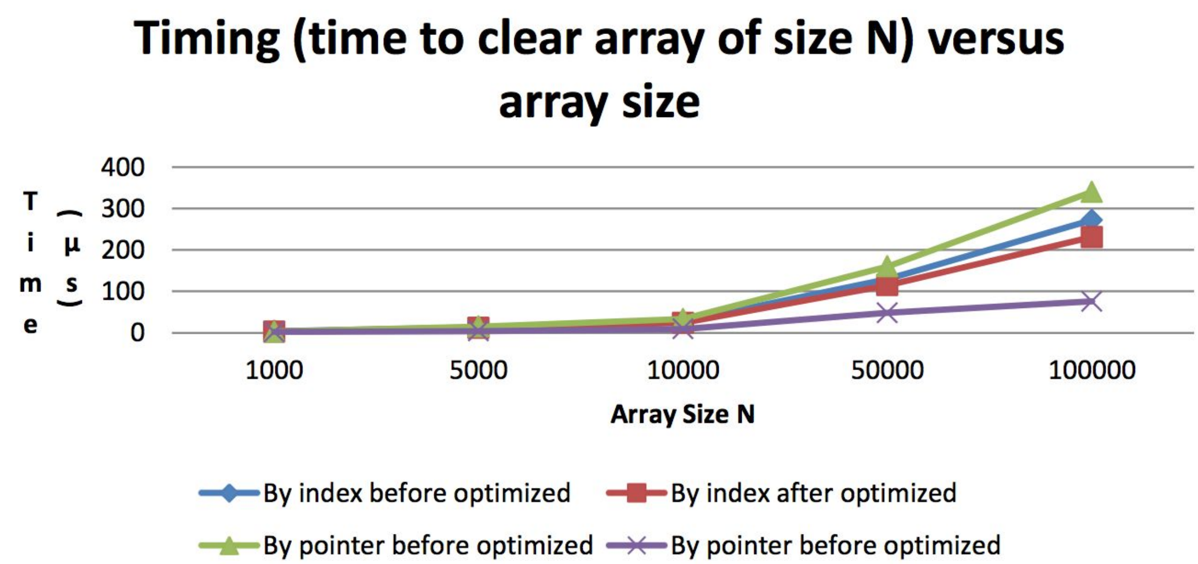
The first is original array n= 1000000, the running time will be 3~4 ms. In the optimization, the running time is less than previous, only 1~2 ms.

```
size: 1000000
Run Time:4.49929ms
size: 1000000
Run Time:3.80481ms
size: 1000000
Run Time:3.72556ms
size: 1000000
Run Time:4.20052ms
size: 1000000
Run Time:3.65485ms
```

```
size: 1000000
Run Time:2.39191ms
size: 1000000
Run Time:1.92949ms
size: 1000000
Run Time:1.07308ms
size: 1000000
Run Time:1.05027ms
size: 1000000
Run Time:2.21971ms
```

❖5. Running Time

As you can see, the larger array size will cost lots of time . As has been mentioned below, four lines which are index before optimized, index after optimized, pointer before optimized and pointer before optimized. The pointer before optimized will increase more faster than other three while they are located in the turning point n=10000.



❖6. Conclusions

In the conclusion, I learned how to debugging in MS Visual Studio and GCC in Linux. This take home test is going to run/debug the optimization. We should to compare the similarity and difference of all threes. That cause each debug system, whatever the Windows X32 bit or Linux X64 bit will debugging or compiling each function. In the end, this assignment required to find out the cost time of each factorial. access members of array like this : for(int i=0; i<n; i++) nArray[i]=nSomeValue; but instead, you access with the following way: for(int* ptrInt = nArray; ptrInt< nArray+n; ptrInt++) *ptrInt=nSomeValue. I spent more than 10 hours to finish all the sections. In the suggestion, it is better to do the efficiently work in less time.