# Analysis the performance of pointers and variables in memory accessing using a programming language on various architectures

**Mr. Rinku[1] Assistant Professor, Mr. Sushil Kumar Bansal[2] Associate Professor**
**Computer Science and Engineering Department,**
**Chitkara University, Baddi,**
**Solan, Himachal Pradesh.**

## Abstract:

Now a day's all high level programming languages are object oriented. All of the languages make use of variables to store data that is volatile. Some of the programming language uses pointers to refer the data, objects or functions etc. Both variable and pointers can be use in various operations of the programming language that will access the memory. Here we have implemented a c/c++ programming to analyze the performance of variables and pointers on various Intel, AMD architectures in accessing the memory. The program that shows the accessing mechanism of a pointer as well as a variable to access memory and verify the performance on various architectures. This paper represents how a pointer variable differ in performance as compared to a variable simply defined in memory access and shows the dependency on the processing speed of the architecture and various other factors.

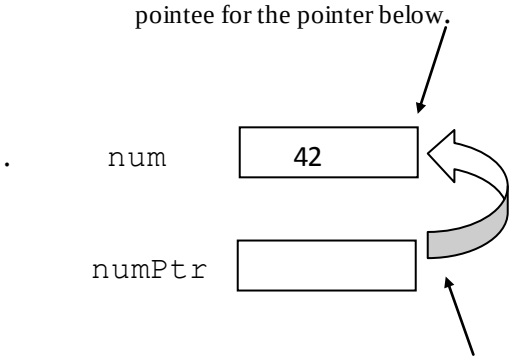
Keywords: Pointers, functions, AMD, Dependency, memory, architectures

## Introduction to pointer and memory

There's a lot of nice, tidy code you can write without knowing about pointers. But once you learn to use the power of pointers, you can never go back. There are too many things that can only be done with pointers. But with increased power comes increased responsibility. Pointers allow new and uglier types of bugs, and pointer bugs can crash in random ways which makes them more difficult to debug. Nonetheless, even with their problems, pointers are an irresistibly powerful programming construct. Pointers solve two common software problems. First, pointers allow different sections of code to share information easily. You can get the same effect by copying information back and forth, but pointers solve the problem better. Second, pointers enable complex "linked" data structures like linked lists and binary trees. Pointers are of various types depending On their word size in memory to store address like far, near, huge, void, null pointer. Each type of pointer has their own specialization. Simple int and float variables operate pretty intuitively. An int variable is like a box which can store a single int value such as 42. In a drawing, a simple variable is a box with its current value drawn inside.


A pointer works a little differently— it does not store a simple value directly. Instead, a pointer stores a reference to another value. The variable the pointer refers to is sometimes known as its "pointee". In a drawing, a pointer is a box which contains the beginning of an arrow which leads to its pointee. (There is no single, official, word for the concept of a pointee — pointee is just the word used in these explanations.) The following drawing shows two variables: num and numPtr. The simple variable num contains the value 42 in the usual way. The variable numPtr is a pointer which contains a reference to the variable num. The numPtr variable is the pointer and num is its pointee. What is stored inside of numPtr? Its value is not an int. Its value is a reference to an int.

A simple int variable. The current value is the integer 42. This variable also plays the role of pointee for the pointer below.




A pointer variable. The current value is a reference to the pointee num above.

Programming Language like C provides a mechanism to determine the address of a specific variable. Remember that if a variable spans multiple bytes, its address is the first byte it occupies. This is done with a special operator that is called the address-of operator, and is represented by the symbol '&'. To clearly illustrate the use of the address-of operator, let's see a minimal code fragment that uses it to print the address of a variable.

```
int num;
int *numPtr;
numPtr=&num;
printf ("The address is: %u\n", p);
printf ("The value is: %d\n",*p);
```

Two pointers which both refer to a single pointee are said to be "sharing". That two or more entities can cooperatively share a single memory structure is a key advantage of pointers in all computer languages. Pointer manipulation is just technique sharing is often the real goal.

## Memory and Variables

Local variables are the programming structure everyone uses but no one thinks about. You think about them a little when first mastering the syntax. But after a few weeks, the variables are so automatic that you soon forget to think about how they work. This situation is a credit to modern programming languages— most of the time variables appear automatically when you need them, and they disappear automatically when you are finished. For basic programming, this is a fine situation. Variables represent storage space in the computer's memory. Each variable presents convenient names like length or num in the source code. Behind the scenes at runtime, each variable uses an area of the computer's memory to store its value. It is not the case that every variable in a program has a permanently assigned area of memory. Instead, modern languages are smart about giving memory to a variable only when necessary. The terminology is that a variable is allocated when it is given an area of memory to store its value. While the variable is allocated, it can operate as a variable in the usual way to hold a value. A variable is deallocated when the system reclaims the memory from the variable, so it no longer has an area to store its value. For a variable, the period of time from its allocation until its deallocation is called its lifetime. The most common memory related error is using a deallocated variable. For local variables, modern languages automatically protect against this error.

The most common variables you use are "local" variables within functions such as the variables num and result in the following function. All of the local variables and parameters taken together are called its "local storage" or just its "locals", such as num and result in the following code...

```
// Local storage example
int Square (int num) {
int result;
result = num * num;
return result;
}
```

The variables are called "local" to capture the idea that their lifetime is tied to the function where they are declared. Whenever the function runs, its local variables are allocated. When the function exits, its locals are deallocated. For the above example, that
means that when the Square () function is called; local storage is allocated for num and result. Statements like result = num * num; in the function use the local storage. When the function finally exits, its local storage is deallocated.

Here is a more detailed version of the rules of local storage...

1. When a function is called, memory is allocated for all of its locals. In other words, when the flow of control hits the starting '{' for the function, all of its locals are allocated memory. Parameters such as num and local variables such as result in the above example both count as locals. The only difference between parameters and local variables is that parameters start out with a value copied from the caller while local variables start with random initial values. This article mostly uses simple int variables for its examples, however local allocation works for any type: structs, arrays. These can all be allocated locally.

2. The memory for the locals continues to be allocated so long as the thread of control is within the owning function. Locals continue to exist even if the function temporarily passes off the thread of control by calling another function. The locals exist undisturbed through all of this.

3. Finally, when the function finishes and exits, its locals are deallocated. This makes sense in a way suppose the locals were somehow to continue to exist — how could the code even refer to them? The names like num and result only make sense within the body of Square() anyway. Once the flow of control leaves that body, there is no way to refer to the locals even if they were allocated. That locals are available ("scoped") only within their owning function is known as "lexical scoping" and pretty much all languages do it that way now.

## The Size of a Variable or pointer

We can easily determine how much space a variable or a pointer occupies, with the sizeof operator. For example, on my computer, sizeof(x) gives me the value 4. This means that an integer needs 4 consecutive bytes in memory to be stored.

## Related Work

In this paper we have implemented a general algorithm that is to be implemented in C/C++ Programming Language on Turbo C++ IDE/Code Block to compare the processing performance whenever a variable and pointer tries to access the same memory location or variable memory location in continuous or non-continuous manner from the primary memory or from the secondary memory.

General Algorithm:

- Import Library files
- Define a function
- Define variables for time to start and stop the processing.
- Define clock variable to correlate the system clock.
- Start the time.
- Start a loop for more than 10,000 counts to find the visible difference.
- Access a same memory location or different memory location using variable or pointer.
- Start and continue the iteration until loop ends.
- Stop time function and print result.

The above general algorithm can implement using C/C++ Programming Language on Turbo C++ IDE or any other IDE like Code Block. The example code snippet is given below that will implements a pointer. In given example pointer access the same memory location.

Pseudo-code for Above Algorithm
Import/include Library Files
Start Main ()
Time Start, Stop
DataType Temp, Variable/Array
Start (Time)
Loop (variable<15000) do
Statement operation of variable/pointer
Increment in loop variable
End Loop
Show the result
End Main ()

The given Algorithm or pseudo code is implemented using a sample program written in C / C++ language executed on TurboC++ IDE/ Code Block IDE...

```
#include <time.h>
#include <stdio.h>
int main(int argc, char *argv[]) {
  time_t start, stop;
  clock_t ticks; long count;
  long int i=0;
  int a,*p;
  a=10;
  time(&start);
  while(i<15000)
  {
        p=&a;
        printf("Work work %d number = %ld\n", *p,i);
        i++;
        ticks = clock();
  }
  time(&stop);
  printf("Used %lf seconds of CPU time. \n", (double)ticks/CLOCKS_PER_SEC);
  printf("Finished in about %.0f seconds. \n", difftime(stop, start));
  getch();
  return 0;
}
```

The same general algorithm can be implemented using a simple variable. The example code snippet is given below.

```
#include <time.h>
#include <stdio.h>
int main(int argc, char *argv[]) {
  time_t start, stop;
  clock_t ticks; long count;
  long int i=0;
  int a,*p;
  time(&start);
  while(i<15000)
  {
   a=5;
   printf("Work work %d number= %ld\n", a,i);
   i++;
   ticks = clock();
  }

  time(&stop);
  printf("Used %lf seconds of CPU time. \n", (double)ticks/CLOCKS_PER_SEC);
```

```
  printf("Finished in about %.0f seconds. \n", difftime(stop, start));
  getch();
}
```

The above code snippet is implemented on various architectures of Intel; AMD etc.This algorithm is implemented on Intel core 2 duo, Intel core i3, and Intel Pentium D, Intel Pentium N Series Celeron and AMD processors.

## Result Analysis

On implemented the above scenario we get various results that shows that the pointer access the memory as fast as compared to the variable accession of memory. The given table shows the memory access results shows on various computer architectures.

| Architectures | Times used by Pointer(in µs/ns) | Time used by variable(in µs/ns) |
|---|---|---|
| Intel Core i3 | 0.069824 | 0.069800 |
| Intel Core 2 Duo | 0.164835 | 0.164782 |
| Intel Dual Core | 0.297846 | 0.486548 |
| Intel Pentium D | 0.494505 | 0.384615 |
| Intel Pentium 4 | 0.384615 | 0.659341 |
| Intel Celeron | 1.989011 | 3.021978 |
| AMD Athelon | 0.398746 | 0.576948 |

Table 1

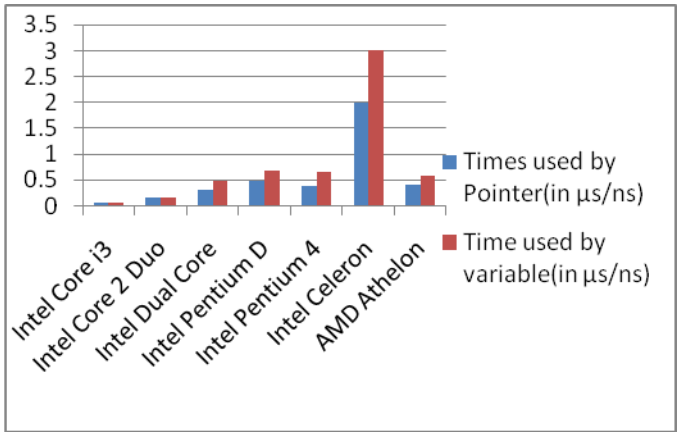



Fig. 1

The given result checked on various architectures of Intel and AMD processor having variation in memory size, storage capacity, processing speed, and load on processor etc. So the data may vary from architecture to architecture.

The ratio of performance in accessing the memory can be shown as

$$Rm = Np / Nv$$

Where Rm represents the ratio, Np represents the no of micro/nano seconds taken in execution of program using pointer variable and Nv shows the no of micro/nano seconds taken by the variable.

Here one consideration has been made that except the memory accessing part all other sequence of code take equal amount of time in case of pointer or a simple variable.
In getting the above the things that are not to be considered are as follows:

- Size of RAM (Random Access Memory Used).
- Size of Cache Memory.
- Processing Speed of microprocessor.
- Size of Secondary Storage
- Processing Speed of Secondary Memory
- Size of memory left by the operating system or any other system software that is to be used by program. So that page fault should not occur.
- Number of process that are executing on the same time when we execute the program.
- Load Status on microprocessor.
- Clocking speed microprocessor
- Programming algorithm and flow of program

These are some basic factor on which the performance of the pointer and a simple variable depend in accessing the primary or secondary memory using a programming language

## Conclusion

Both pointer and variable can be used in programming to develop software. Pointer has some benefit as well as some deficiency also. Overall pointer directly accessible anywhere using references. They are fast to access the data stored in memory or in any continuous or non-continuous memory location viz. array, Linked List as compared to variable in accessing a memory variable.

## References:

1. Kanetkar Yashwant, Let us C, Array and pointers, 7th edition, BPB publication
2. Tsiombikas John, Pointer Explained
3. Balagurusami E., ANSI C, Pointer Explained, 3rd edition.
4. Ritchie Denis M, C Programming Language, ANSI C, Prentice Hall Software Series.
5. Zhongxing Xu,Ted Kremenek and Jian Zhang, A Memory Model for Static Analysis of C Programs
6. Frank Ch. Eigler, Pointer Use Checking for C/C++, Red Hat
7. Parlante Nick, Pointers and Memory