****

**Dr. Hasan Mahmood Aminul Islam (DHMAI)**

**Assistant Professor, Dept of CSE  
East West University, Bangladesh**  
Specialist System-on-Chip Software (5G) Nokia  
Nokia Headquarters, Espoo, Finland  
D.Sc in Technology (CSE)  
Aalto University, Finland  
M.Sc in Computer Science  
University of Helsinki, Finland  
B.Sc in CSE, BUET, Bangladesh  
Web Page: <https://www.ewubd.edu/faculty-profile/mahmood.aminul>

Personal Website: <https://hmaislam.github.io/myweb/>

**Contact Person:**

| **Md. Farhad Billah**  **Student, Department of CSE**  **East West University, Dhaka, Bangladesh**  **Email: farhadbillah2020@gmail.com** | **Mohua Akter**  **Student, Department of CSE**  **East West University, Dhaka, Bangladesh**  **Email: 2022-3-60-112@std.ewubd.edu** |
| --- | --- |

**Variables**

A variable in a program is something with a name, the value of which can vary. The way the compiler and linker handles this is that it assigns a specific block of memory within the computer to hold the value of that variable.

The size of that block depends on the range over which the variable is allowed to vary. For example, on 32 bit PC's the size of an integer variable is 4 bytes. On older 16 bit PCs integers were 2 bytes. In C the size of a variable type such as an integer need not be the same on all types of machines. We have integers, long integers and short integers which you can read up on in any basic text on C. This document assumes the use of a 32 bit system with 4 byte integers.

**When we declare a variable we inform the compiler of two things, the name of the variable and the type of the variable.**

* Variables in C are used to access a single location of memory and the size of the memory depends on the size of data types.
* Therefore, a variable MUST be declared before using it. When you declare a variable, the compiler automatically allocates memory for that variable.

**Example:**

**data\_type variable\_name;**

**When Loops/Iteration of Some Pieces of Tasks arise, no worries!**

Whatever we start or are heading to work on something new in programming, remember that “Try” to understand the semantics of the new thing and how to use it in your program, what are inherent things you need to capture into your memory.

Remember that nothing needs to be memorized. If you feel that you need to memorize something, then definitely you have some gaps in your logic. If you are not able to find your gaps, that means you really need to consult with your friends and teachers.

**LooK at the loops:** See if you find any difference in logic



**Syntax:**

**Same purpose**





**Where can we use loops?**

* E.g.  **H.S.C / S.S.C Mathematics : সমান্তর ধারা, গুণোত্তর ধারা**
* **Some logic:**
* **Condition**  false ; if value is zero;

true ; otherwise ;

* **for/while** → iteration number: zero or more times



* **do-while** → iteration number: once or more
* **Variables vs Arrays vs Pointers:**
* **Variables**

A variable in a program is something with a name, the value of which can vary. The way the compiler and linker handles this is that it assigns a specific block of memory within the computer to hold the value of that variable. The size of that block depends on the range over which the variable is allowed to vary. For example, on 32 bit PC's the size of an integer variable is 4 bytes. On older 16 bit PCs integers were 2 bytes. In C the size of a variable type such as an integer need not be the same on all types of machines. We have integers, long integers and short integers which you can read up on in any basic text on C. This document assumes the use of a 32 bit system with 4 byte integers. When we declare a variable we inform the compiler of two things, the name of the variable and the type of the variable.

* Variables in C are used to access a single location of memory and the size of the memory depends on the size of data types. Therefore, a variable MUST be declared before using it. When you declare a variable, the compiler automatically allocates memory for that variable.
* In C programming, one of the frequent problems is to handle similar types of data. For example: if the user wants to store marks of 10000 students, this can be done by creating 10000 variables individually but, this is rather tedious and impracticable. These types of problems can be handled in C programming using arrays.
* Arrays can be considered as the collection of variables where the base address of the array is the first element of the array. An array in C Programming can be defined as a number of memory locations, each of which can store the same data type and which can be referenced through the same variable name.
* Arrays and pointers are synonymous in terms of how they are used to access memory. But, the important difference between them is that a pointer variable can take different addresses as values whereas, in the case of an array it is fixed. In C, the name of the array always points to the first element of an array.
* An array in C Programming can be defined as a number of memory locations, each of which can store the same data type and which can be referenced through the same variable name.
* **Important thing to remember in C arrays**

Suppose, you declared the array of 10 students. For example: students[10]. You can use array members from student[0] to student[9]. But, what if you want to use element student[10], student[100] etc. In this case the compiler may not show error using these elements but, may cause fatal error during program execution

**Multidimensional Arrays**

Just consider **each box** is the storage of **a simple variable**. Now you can map the logic of a variable and an element of Array in Programming.

| array | Col - 0 | Col -1 | Col -2 | Col -3 | Col -4 | Col -5 |
| --- | --- | --- | --- | --- | --- | --- |
| Row- 0 | array[0][0] | array[0][1] | array[0][2] | array[0][3] | array[0][4] | array[0][5] |
| Row- 1 |  | array[1][1] |  | array[1][3] |  |  |
| Row- 2 |  |  |  |  |  | array[2][5] |

When we declare a variable we inform the compiler of two things, the name of the variable and the type of the variable. For example, we declare a variable of type integer with the name k by writing:

**int k;**

On seeing the "int" part of this statement the compiler sets aside 4 bytes of memory to hold the value of the integer. It also sets up a symbol table. In that table it adds the symbol k and the relative address in memory where those 4 bytes were set aside.

**k=2**

We expect that, at run time when this statement is executed, the value 2 will be placed in that memory location reserved for the storage of the value of k. In C we refer to a variable such as the integer k as an "object".

In C when we define a pointer variable we do so by preceding its name with an asterisk. We also give our pointer a type which, in this case, refers to the type of data stored at the address we will be storing in our pointer. For example, consider the variable declaration:

int \*ptr;

ptr is the name of our variable, the '\*' informs the compiler that we want a pointer variable, i.e. to set aside however many bytes is required to store an address in memory. The int says that we intend to use our pointer variable to store the address of an integer. Such a pointer is said to "point to" an integer. A pointer initialized in this manner is called a "null" pointer.

The actual bit pattern used for a null pointer may or may not evaluate to zero since there is no value assigned to it. Thus, setting the value of a pointer using the NULL macro, as with an assignment statement such as ptr = NULL, guarantees that the pointer has become a null pointer.

Suppose now that we want to store in ptr the address of our integer variable k. To do this we use the unary & operator and write:

**ptr = &k;**

The "dereferencing operator" is the asterisk and it is used as follows:

**\*ptr = 7;**

**Arrays and Pointers**

Arrays and pointers are closely related in C. In fact an array declared as **int A[10];**

can be accessed using its pointer representation. The name of the array A is a constant pointer to the first element of the array. So A can be considered a const int\*. Since A is a constant pointer, A = NULL would be an illegal statement. Arrays and pointers are synonymous in terms of how they are used to access memory.

But, the important difference between them is that a pointer variable can take different addresses as values whereas, in the case of an array it is fixed.

Consider the following array:

**Declaration: int std[5];**

**std[0] std[1] std[2] std[3] std[4]**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |

In C , **the name of the array** always points to the **first element of an array**. Here, the address of the first element of an array is **std[0]**. Also, **std** represents the address of the pointer where it is pointing. Hence, **&std[0]** is equivalent to **std.** Note, the value inside the address **&std[0]** and address age are equal. Value in address **&std[0]** is **std[0]** and value in address std is **\*std.** Hence, **std[0]** is equivalent to **\*std.**

C arrays can be of any type. We define an array of ints, chars, doubles etc. We can also define an array of pointers as follows. Here is the code to define an array of n char pointers or an array of strings.

**char\* array[n];**

each cell in the array A[i] is a char\* and so it can point to a character. Now if you would like to assign a string to each A[i] you can do something like this.

**array[i] = malloc(length\_of\_string + 1);**

Again this only allocates memory for a string and you still need to copy the characters into this string. So if you are building a dynamic dictionary (n words) you need to allocate memory for n char\*’s and then allocate just the right amount of memory for each string.

| Pointer | Array |
| --- | --- |
| **A pointer is a place in memory that keeps address of another place inside** | **An array is a single, pre allocated chunk of contiguous elements (all of the same type), fixed in size and location** |
| **A pointer is a place in memory that keeps address of another place inside** | **Expression a[4] refers to the 5th element of the array a.** |
| **Pointer can’t be initialized at definition** | **Arrays can be initialized at definition. Example**  **int num[] = { 2, 4, 5}** |
| **Pointer is dynamic in nature. The memory allocation can be resized or freed later.** | **They are static in nature. Once memory is allocated , it cannot be resized or freed dynamically** |

**(MMAP) Variables vs Arrays vs Pointers:**















Type: unsigned/signed number





**Unsigned data type:** uint8\_t

uint16\_t

uint32\_t

8/16/32 bits

Mapping: 

**A single location**

**Multiple addresses** but in the case of Array the first address is randomly allocated and the rest addresses will be contiguous and the size of the memory will be based on your declaration for the array. Again note that the first address, index 0 (zero) will be allocated randomly by the compiler (which is called the base address) and the rest address will follow contiguously of the base address.





0 1 … … n

Figure: Each box represents one memory address whose size is defined by the data type of the Array. The figure has been drawn in this way to dictate that an array is nothing but a collection of variables.

* **Starting to think like a C programmer**

**Now that we have spent some time studying and tried to understand the core of C programming, we are talking about the C language. You may have been trying to think like a Java programmer and convert that thought to C. Now it is time to think like a C programmer. Being able to think directly in C will make you a better C programmer.**

* **Here are 15 things to remember when you start a C program from scratch.**

1. include <stdio.h> and all other related headers in all your programs.
2. Declare functions and variables before using them
3. Better to increment and decrement with ++ and -- operators.
4. Better to use x += 5 instead of x = x +5
5. A string is an array of characters ending with a ‘\0”. Don’t ever forget the null character
6. Array of size n has indices from 0 to n-1. Although C will allow you to access A[n] it is very dangerous
7. A character can be represented by an integer (ASCII value) and can be used as such
8. The unary operator & produces an address
9. The unary operator \* dereference a pointer
10. Arguments to functions are always passed by value. But the argument can be an address of just a value
11. For efficiency, pointers can be passed to or return from a function
12. Logical false is zero and anything else is true
13. You can do things like for(;;) or while(i++) for program efficiency and understanding
14. Use /\* .. \*/ instead of //, it makes the code look better and readable
15. The last and most important one, always compile your program before submitting or showing to someone. Don’t assume that your code is compilable
16. and contains no errors. Try using –std=c99, which is the c99 standard. It's better. (Although, c11 is also on its way but not a standard at the moment for all machines)