POINTER IN C / C++

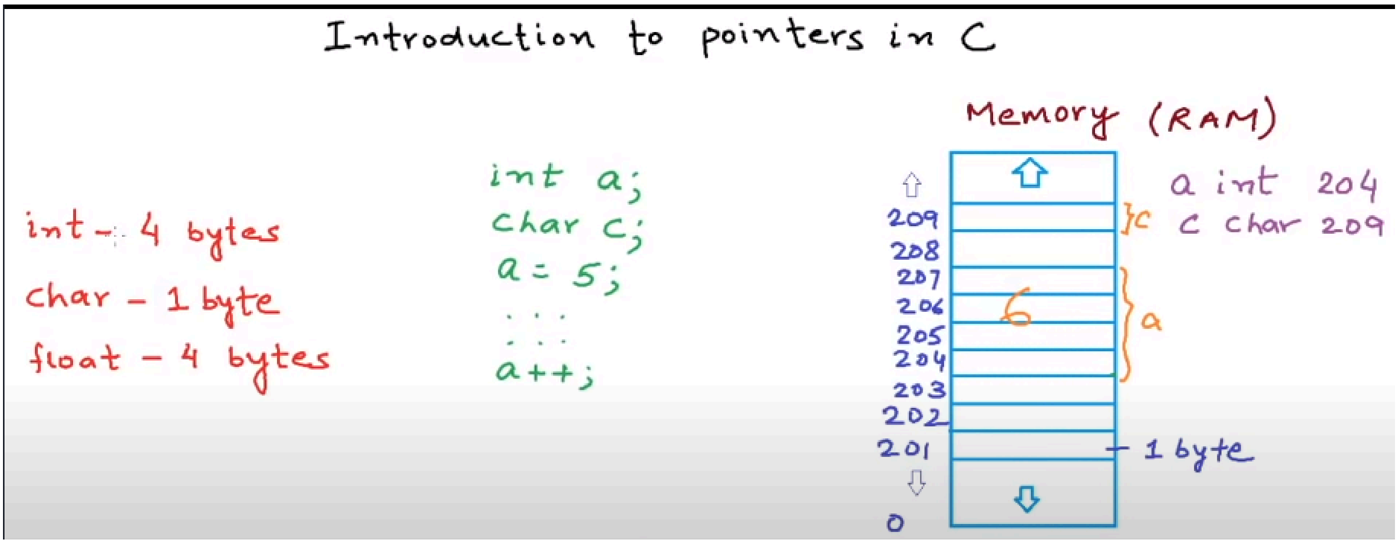
Table of Contents

Pointers in C / C++

Pointers are variables that store the address of another variable.

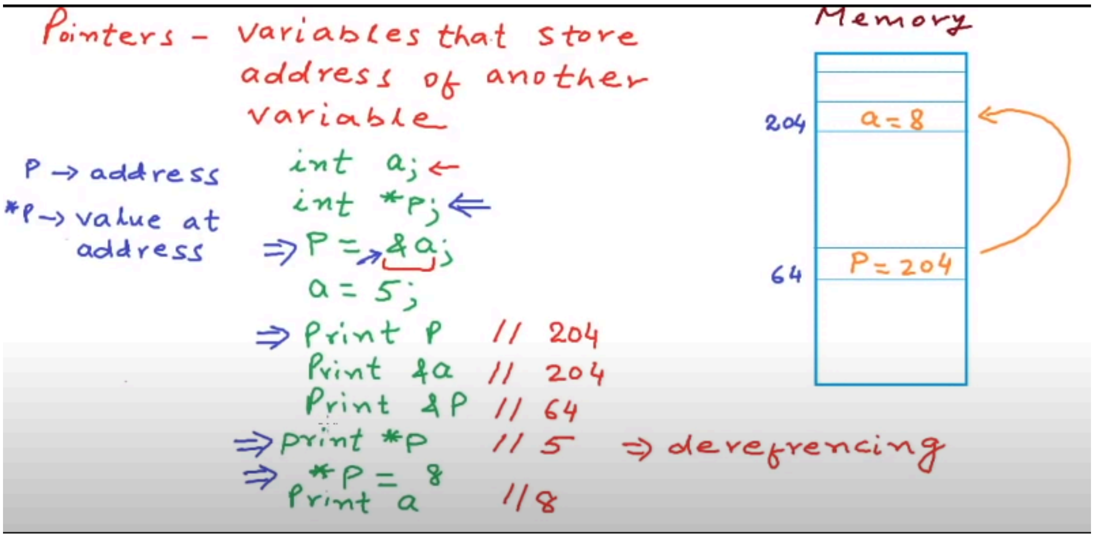
1. Introduction to pointers in C/C++

When we declare a variable in our program, for example, if we declare a variable of type integer, then when this program executes, the computer allocates some amount of memory corresponding to this particular variable. How much memory it allocates, depends upon the data type, and also upon the compiler. So in a typical modern-day compiler, an integer is allocated four bytes of memory. Character variable is allocated one byte of memory, float is allocated four bytes of memory and we can have other variables as well. As soon as the computer sees a declaration, like this, during the program's execution, it knows that this is an integer variable, so we must allocate four bytes of memory. Let's say in our example, it allocates memory starting address 204 and ending address 207 to a. And the computer has an internal structure, a lookup table, where it stores this information that there is a variable a, it is of type integer, and it is located at address 204, which is the starting address of the variable. Now, if we declare another variable for example, if we declare a variable named C, which is of type character, once again, when the machine sees this declaration, it knows that it is a character variable. Hence, it needs one byte of memory. So it looks for some free space, let's say in this case, it allocates the address to 209 and 2094 bytes to C, and once again, it keeps an entry for it in an internal structure called a lookup table, that sees a character and its addresses 209. Now when we perform some operation with these variables, like let's say if we initialize a to five, when our machine or computer sees such a statement, it looks into the lookup table for this variable A. So it finds this variable a that it is an integer and it is at address 204. So, it goes at address 204 and in these four bytes starting 204 it writes this value five now in reality, the value is written in binary but for the sake of understanding, we are writing here in decimal form. Now once again, let's say we have some statements and then again after these statements, we have another statement which increments a Now again, when the computer sees that, he has to be incremented, it again looks for this address for a goes to the address and modifies this value at this particular address. So this block of memory allocated for a store the value six now. Now, all of this is cool, but can we know the address of a variable



2. Working with Pointers

we have a block of four bytes at address two sorts of food that stores an integer variable a. Now we can have another variable, the type of which is a pointer to an integer. And let us say the name of this variable is P. Now this variable p can store the address of A. And using the properties of P, or using some operators upon p, we can reach a. Now p also takes some memory. So let's say it is stored at location address 64. And it also takes four bytes of memory, we can also modify p to point to another integer. So let's say we have another integer at address 208, named B, and having value 10. And if we change the address in p from 204 to 208, then P now points to me.



2.1. Example 1

```
/*******************************************
*******************************************
   Introduction to pointers in C/C++
*******************************************
*******************************************/
int a;                          //creating a variable of type int
int *p;                         //creating a pointer of type int
p = &a;                         //assigning  p the address of a
std::cout << "initial value:    " << a << std::endl; //garbage value in a
a = 5;                          //assigning  a value 5
std::cout << "pointer:  " << p << std::endl;
std::cout << "address to value: " << &a << std::endl;
std::cout << "address to pointer: " << &p << std::endl;
std::cout << "pointer to value: " << *p << std::endl;
*p = 8;                         //accessing a through pointer p
std::cout << "value changed via pointer:    " << a << std::endl;
int b = 100;
*p = b;
std::cout << "value of a change with pointer:   " << a << std::endl;
```

Output

```
initial value:  21905
pointer:        0x7fffe42ff0ec
address to value: 0x7fffe42ff0ec
address to pointer: 0x7fffe42ff0f8
pointer to value:       5
value changed via pointer:      8
value of a change with pointer: 100
```

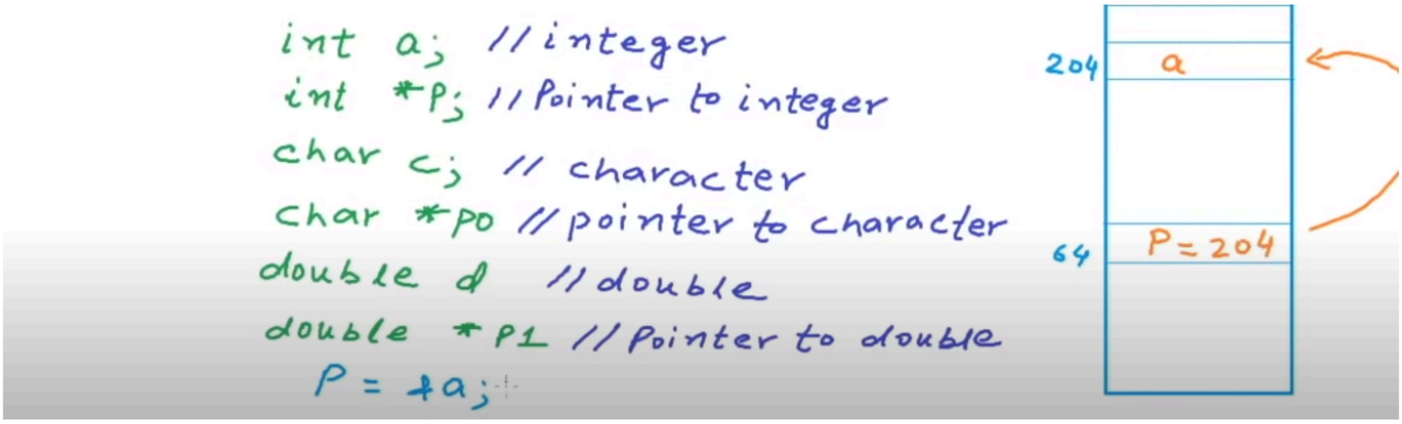
2.2. Example 2

```
/*******************************************
******************************************
      Arithmetic operation on pointer
******************************************
******************************************/
int a = 10;
int *p;
p = &a;
std::cout << "pointer:      " << p << std::endl;
std::cout << "pointer increment:    " << p + 1 << std::endl;// difference of p+1 and p is 4 as p is of type int
std::cout << "step increment:       " << sizeof(a) << std::endl;
std::cout << "value at *(p+1):      " << *(p + 1) << std::endl; //some garbage value
```

Output

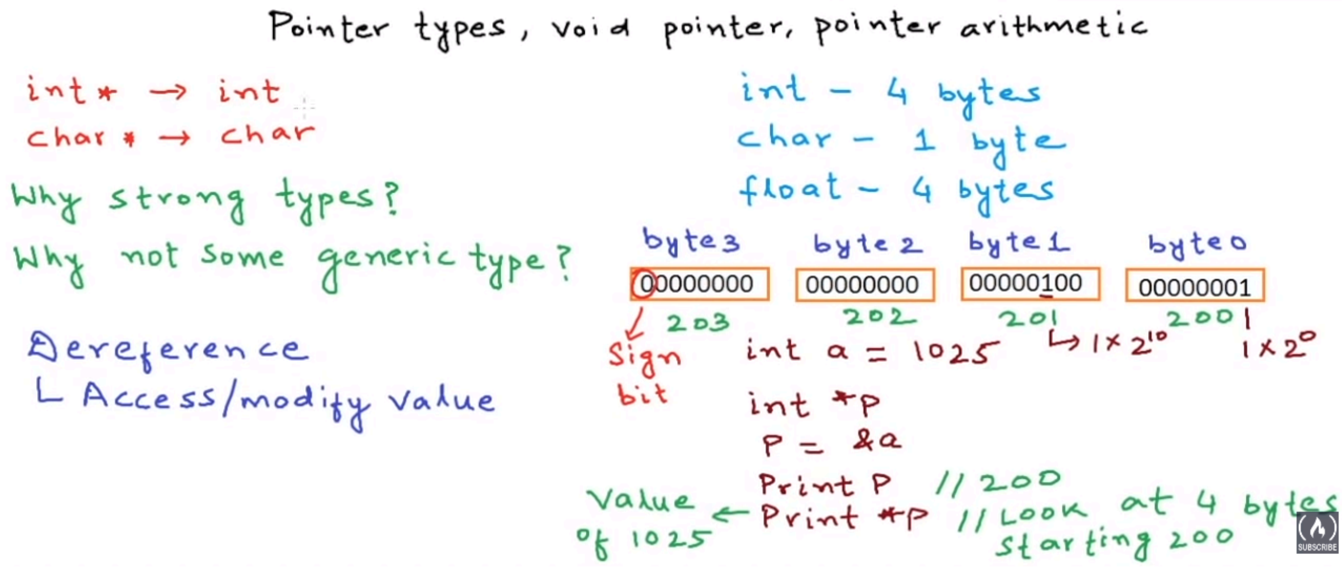
```
pointer:                0x7ffea3770e28
pointer increment:      0x7ffea3770e2c
step increment:         4
value at *(p+1):                0
```

3. Pointer types, Pointer arithmetic, void pointers



pointer variable of a particular type to store the address of a particular type of variable.
So int stores a pointer to an integer will be needed to store the address of an integer character pointer will be needed to store that does have a character. Similarly, if we have a user-defined structure or class, then we need a pointer of that particular type only.
But why do we need these strong types? Isn't it that the pointer variables just store the address of the variable? So why couldn't we have just one type? That will be some generic type to store the address of all kinds of variables.
And the answer is that we do not use the pointer variables only to store memory addresses. But we also use them to dereference these addresses so that we can access and modify the values in these addresses.
Now, as we know, data types have different sizes, like in a typical modern-day compiler and integer, it's stored in four bytes. A character variable is stored in one bite of the float and is again stored in four bytes. These variables differ not only in their sizes but also in how we store information in whatever bytes are available for these variables or data types. Let's say we have an integer a and its value is 1025. And this is how it is laid out in the memory each bracket here is one byte.
So let's say this particular byte which is the least significant byte is byte zero, and then we go on like bite one, bite two, and by three. Now we also know that each byte in the memory is addressable. Let's say the address of bytes zero is 200. Now these four bytes need to be contiguous let's say the address of byte to byte one is 201. And then we go on like 202 and 203. When an integer is represented in the memory is stored in the memory the leftmost bit stores the information on whether this integer is positive or negative, so this is also called a signed bit, signed bit, and the remaining 31 bits are used to store the value. So if you see we have a one at right most bet with a place value two to the power zero and at this particular bit with a place value to to the About 10. So the overall value that we have in binary here is one zero to five in decimal. Now, what if I declare a pointer to integer P and store the address of A in p by using the ampersand operator, what will happen if I print the value of p, the value of p or the address stored in p be 200, the address of bytes zero. So we're kind of saying that we have the address of an integer variable starting at address 200. If we dereference this address and try to print *P, we want to know the value at this particular address, then the machine sees that okay, P is a pointer to an integer, so we need to look at four bytes starting address 200. Then the machine knows how to extract and extract the value of an integer data type. So it retrieves the value one zero to five out of these four bytes.
Now, if p was a character pointer, then while dereferencing, the machine would have looked at only one byte because a character variable is only one byte. If P was appointed to float, then although float is also stored in

four bytes, the way information is written for float in these four bytes is different from the way information is returned for an integer datatype. So the value printed would have been something else.



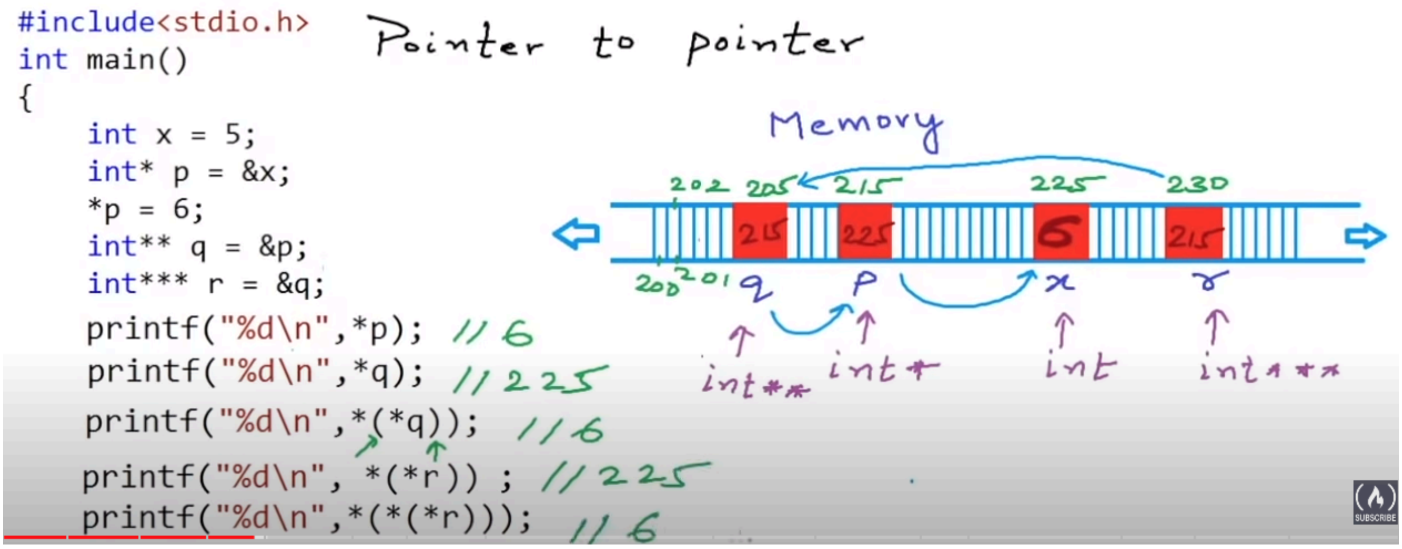
3.1. Example 1

```
/*******************************************************
********************************************************
Pointer Types, Pointer Arithmetic , void pointer
********************************************************
********************************************************/
int a = 1025;
int *p = &a;
std::cout << "size of integer:  " << sizeof(int) << " bytes" << std::endl;
std::cout << "address pointing to a  :  " << p << " value of pointer:   " << *p << std::endl;
std::cout << "address pointing to p+1:  " << p+1 << "   value of pointer:   " << *(p+1) << std::endl;
// this increment is equal to 4 bytes as p stores sizeof(int) bytes p+1 address would be p's address + 4bytes
char * p0;
p0 = (char * )p; //type cast
std::cout << "size of char " << sizeof(char) << std::endl;
std::cout << "Address of p0:    " << (void*)p0 << " value of p0:    " <<(void*)(*p0) << std::endl;
std::cout << "Address of p0:    " << (void*)p0+1 << " value of p0+1:    " <<(void*)(*(p0+1)) << std::endl;
//base address of pointer to integer is shored in P0 so P0 point first byte of p that is 1
//p0 +1 would point to second byte of p that is 4
//  1025 = 00000000 00000000 00001000 0000001
//  1025 =      0        0        4       1
// generic pointer type
/*
void pointer type  is not mapped to a particular data type,
we cannot dereference this particular pointer  variable.
So, if you try to print *P0 will give you  an error we are
getting a compilation error,
*/
void *P0;
P0=p;
//std::cout << "Address of p0:  " << (void*)P0 << " value of P0:    " <<(void*)(*P0) << std::endl; //compilation error
std::cout << "Address of p0  :  " << (void*)P0 << std::endl; //we only print the address
std::cout << "Address of p0+1:  " << (void*)P0+1 << std::endl; //we only print the address
```

Output

```
size of integer:         4 bytes
address pointing to a  :        0x7ffc5d14f610  value of pointer:       1025
address pointing to p+1:        0x7ffc5d14f614  value of pointer:       32607
size of char 1
Address of p0:  0x7ffc5d14f610 value of p0:     0x1
Address of p0:  0x7ffc5d14f611 value of p0+1:   0x4
Address of p0  :        0x7ffc5d14f610
Address of p0+1:        0x7ffc5d14f611
```

4. Pointers to Pointers in C/C++



4.1. Example

```
/*******************************************
******************************************
        pointer to pointer
******************************************
*******************************************/
int x = 5;
int *p = &x;
std::cout << std::hex << "value of p           " << p << std::endl;
std::cout << std::hex << "address of p         " << &p << std::endl;
std::cout << std::hex << "pointer value p          " << *p << std::endl;
*p = 6;
std::cout << "Updating x though its pointer p" << std::endl;
std::cout << std::hex << "pointer value p          " << *p << std::endl;
std::cout << std::hex << "value of x           " << x << std::endl;
int **q = &p;
std::cout << std::hex << "value of p           " << p << std::endl;
std::cout << std::hex << "pointer value of q       " << *q << std::endl;

std::cout << std::hex << "pointer value of p       " << *p << std::endl;
std::cout << std::hex << "pointer to pointer  of q  " << **q << std::endl;
```

Output

```
value of p                      0x7ffefe45bf1c
address of p                    0x7ffefe45bf28
pointer value p                 5
Updating x though its pointer p
pointer value p                 6
value of x                      6
value of p                      0x7ffefe45bf1c
pointer value of q              0x7ffefe45bf1c
pointer value of p              6
pointer to pointer  of q        6
```

5. Pointers as function arguments - call by reference

5.1. Example

```
/*****************************************************
****************************************************
Pointers as function arguments - call by reference
****************************************************
****************************************************/
int a = 5;
std::cout<<std::hex<<"address of a:             "<<&a<<std::endl;
local_function_increment(a);
std::cout<<std::hex<<"value of a: "<<a<<std::endl;
call_by_reference(&a);
std::cout<<std::hex<<"value of a: "<<a<<std::endl;
```

Function definition

```
void local_function_increment(int x){  //call by value function
    /*
    when function is called new variable is declared
    & value id copied to that variable
    when a function end it clears the stack frame that was located
    for  increment the local variable
    and main method resumes again
    lifetime of a local variable is
    till the time the function is executing.
    */
    x=x+1;
    std::cout<<std::hex<<"address of a local_function_increment :   "<<&x<<std::endl;
}
void call_by_reference(int *p){
    /*
    when you this line execute
        *p=&a
    */
    *p=(*p)+1;
}
```

Output

```
address of a:                             0x7ffd11326e48
address of a local_function_increment : 0x7ffd11326e1c
value of a: 5
value of a: 6
```

6. Pointers and arrays

6.1. Example

```
/****************************************
****************************************
        pointer and Array
****************************************
****************************************/
int A[5]={2,4,5,8,1};
int *p=&A[0];
std::cout<<"address of A              "<<&A<<std::endl;
std::cout<<"Value of P (address of A[0])    "<<p<<std::endl;
/* base address of A array) is same as address of A[0]*/
std::cout<<"pointer value p     "<<*p<<std::endl;
std::cout<<"value of A[0]       "<<*A<<std::endl;
std::cout<<"pointer value p+1   "<<*(p+1)<<std::endl;
std::cout<<"value of A[1]       "<<*(A+1)<<std::endl;
std::cout<<"pointer value p+2   "<<*(p+2)<<std::endl;
std::cout<<"value of A[2]       "<<*(A+2)<<std::endl;
std::cout<<"pointer value p+3   "<<*(p+3)<<std::endl;
std::cout<<"value of A[3]       "<<*(A+3)<<std::endl;
// A++   ==> would give us error
// after assigning address to pointer we can do p++
```

Output

```
address of A                    0x7ffd76358030
Value of P (address of A[0])    0x7ffd76358030
pointer value p         2
value of A[0]           2
pointer value p+1       4
value of A[1]           4
pointer value p+2       5
value of A[2]           5
pointer value p+3       8
value of A[3]           8
```

7. Arrays as function arguments

7.1. Example

```
/*******************************************
*******************************************
        Array as function argument
*******************************************
*******************************************/
int A[] = {1, 2, 3, 4, 5};
int size = sizeof(A) / sizeof(A[0]);
std::cout << "size of A inside main " << size << std::endl;
int total = sumofElement(A);
std::cout << "sum of element:   " << total << std::endl;
int total_ref = sumofElement_ref(&A[0], size);
std::cout << "sum of element ref:   " << total_ref << std::endl;
Double(A,size);
for (int i = 0; i < size; i++){
    std::cout<<A[i]<<"  ";
}
std::cout<<std::endl;
```

```
int sumofElement(int A[])
{
    /*
    when is called it does not copy the whole array
    it just create a pointer to array
    int A[]  ==> int *A  (both are same here )
    array are always passed as reference to function
    if we dont pass it by reference we cannot
    calculate the correct size of array

    one advantage is that every time we call the function we don't want to
    copy the whole big array
    */
    int sum = 0;
    int size = sizeof(A) / sizeof(A[0]);
    std::cout << "size of A inside function " << size << std::endl;

    for (int i = 0; i < size; i++)
    {
        sum += A[i];
    }
    return sum;
}
```

```
int sumofElement_ref(int *A, int size)
{
    int sum = 0;

    for (int i = 0; i < size; i++)
    {
        sum += A[i];
    }
    return sum;
}
```

```
void Double(int *A, int size)
{
    for (int i = 0; i < size; i++)
    {
        A[i]=2*A[i];
    }

}
```

Output

```
size of A inside main 5
size of A inside function 2
sum of element: 3
size of A ref function 5
sum of A ref:   15
2       4       6       8       10
```

18. References

- [Pointers in C / C++ [Full Course]]

19. Revision History Table

Date	author	Version	Comments
18-01-2024	MMuneeb Tahir	V1	Pointer basics added