Pointer Arithmetic

- When you add to or subtract from a pointer, the amount by which you do that **is multiplied by the size of the type** the pointer points to.
- In the case of our three increments, each 1 that you added was multiplied by sizeof(int).

```
int array[] = { 45, 67, 89 };
int *array_ptr = array;
printf(" first element: %i\n", *(array_ptr++));[1]
printf("second element: %i\n", *(array_ptr++));
printf(" third element: %i\n", *array_ptr);
```

Output:

```
first element: 45
second element: 67
third element: 89
```

NOTE 1: 1==4 (programmer humor?!)
*(array_ptr++) == *array_ptr++

BTW

*(array_ptr++) [1]
vs
(*array_ptr)++

find the value at that address, output, then add "1" to the address
VS
Find the value at the address, output, then add one to the value at that address

Pointer Arithmetic (cont)

Expression	Assuming p is a pointer to a…	… and the size of *p is…	Value added to the pointer
p+1	char	1	1
p+1	short	2	2
p+1	int	4	4
p+1	double	8	8
p+2	char	1	2
p+2	short	2	4
p+2	int	4	8
p+2	double	8	16

Pointer Arithmetic (again)

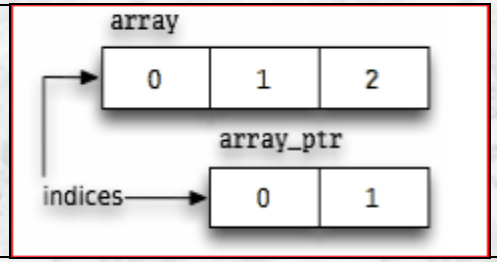
- pointer (+ or -) integer
  - Only for pointers that are pointing at an element of an array
  - Also works with malloc
  - Watch for bounds (begin and end)
    - Ok to go one beyond the array but not a valid dereference
- pointer#1 – pointer#2
  - Only allowed when both point to elements of the same array and p1 index < p2 index
  - Measured in array elements not bytes
  - If p1 → array[i] and p2 → array[j] then p2-p1 == j - i

Pointer Indexing

```
int array[] = { 45, 67, 89 };
printf("%i\n", array[0]);    // output is 45
// array and array[0] point to same thing
```

- The subscript operator (the [] in array[0]) has *nothing to do with arrays*.
- In most contexts, arrays decay to pointers. This is one of them: That's a *pointer* you passed to that operator, not an array.

```
int array[] = { 45, 67, 89 };
int *array_ptr = &array[1];
printf("%i\n", array_ptr[1]);
//output is 89 (whooooooooaaaahhhhtttt??!!)
```



- array points to the first element of the array;
  - **array[1] == *(array + 1)**
- array_ptr is set to &array[1], so it points to the second element of the array.
- So array_ptr[1] is equivalent to array[2]

NULL vs 0 vs '\0'

- NULL is a macro defined in several standard headers
- 0 is an integer constant
- '\0' is a character constant, and
  - nul is the name of the character constant.

All of these are *not* interchangeable

- NULL is to be used for pointers only since it may be defined as ((void *) 0), this would cause problems with anything but pointers.
- 0 can be used anywhere, it is the generic symbol for each type's zero value and the compiler will sort things out.
- '\0' should be used only in a character context.
  - nul is not defined in C or C++, it shouldn't be used unless you define it yourself in a suitable manner, like:
    - #define nul '\0'

NULL pointer and VOID

- 0 (an integer value) is convertible to a null pointer value if assigned to a pointer type
- VOID – no value at all – literally means "nothing"
  - So it is type-less (no type defined) so can hold any type of pointer
  - We cannot perform arithmetic on *void* pointers (no type defined)
  - Cannot dereference (can't say, "get the value at that address" – no type defined)
- NULL is defined as 0 cast to a void * pointer
  - #define NULL (void *) 0;

FYI: However, NULL and zero are not the same as no returned value at all, which is what is meant by a void return value (see your first C program examples)

- Is there any difference between the following two statements?
  char *p=0;
  char *t=NULL;
  NO difference. NULL is *#defined* as 0 in the 'stdio.h' file. Thus, both *p* and *t* are NULL pointers.

- Is this a correct way for NULL pointer assignment?
  int i=0;
  char *q=(char*)i; *// char * cannot point to an int type… even for a moment in time*
  NO. Correct → *char *q=0 (or) char *q=(char*)0*

- Is the NULL pointer same as an uninitialized pointer? NO

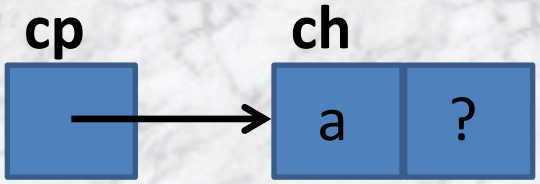
R and L values

- L-value = something that can appear on the left side of an equal sign
  - A place i.e. memory location for a value to be stored
- R-value is something that can appear on the right side of an equal sign
  - A value
- Example:
  - a = b+25 vs b+25 = a
- Example:
  - int a[30];
  - a[b+10]=0;
- Example:
  - int a, *pi;
  - pi = &a;
  - *pi = 20;

R and L values (cont)

- Given:
  - char ch = 'a';
  - char *cp = &ch;

NOTE: the ? is the location that follows ch



Problem	Expression	R-value	L-value
1	ch	yes	yes
2	&ch	yes	illegal
3	cp	yes	yes
4	&cp	yes	illegal
5	*cp	yes	yes
6	*c+1	yes	illegal
7	*(c+1)	yes	yes
8	++cp	yes	illegal
9	cp++	yes	illegal
10	*++cp	yes	yes
11	*cp++	yes	yes
12	++*cp	yes	illegal
13	(*cp)++	yes	illegal
14	++*++cp	yes	illegal
15	++*cp++	yes	illegal

An Array of Character Pointers

```
#include<stdio.h>
int main()
{
    char *ptr1 = "Himanshu";
    char *ptr2 = "Arora";
    char *ptr3 = "TheGeekStuff";

    char* arr[3];

    arr[0] = ptr1;
    arr[1] = ptr2;
    arr[2] = ptr3;

    printf("\n [%s]\n", arr[0]);
    printf("\n [%s]\n", arr[1]);
    printf("\n [%s]\n", arr[2]);
    return 0;
}
```

// Declaring/Initializing 3 characters pointers

//Declaring an array of 3 char pointers

// Initializing the array with values

//Printing the values stored in array

Pointers to Arrays

- <data type> (*<name of ptr>)[<an integer>]
  - Declares a pointer ptr to an array of 5 integers.
    - int(*ptr)[5];

```
#include<stdio.h>
int main(void)
{    char arr[3];
     char (*ptr)[3];
     arr[0] = 'a';
     arr[1] = 'b';
     arr[2] = 'c';
     ptr = &arr;
     return 0;
}
```

Declares and initializes an array ‘arr’ and then declares a pointer ‘ptr’ to an array of 3 characters. Then initializes ptr with the address of array ‘arr’.

```
int *arr[8];       // An array of int pointers.
int (*arr)[8];     // A pointer to an array of integers
```