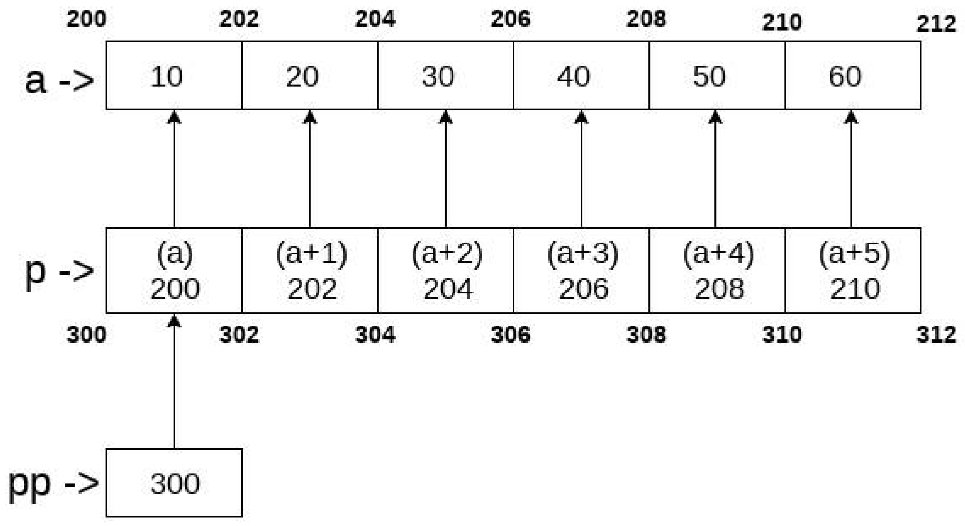
Tut. Sheet 3

Solutions @Ayush Rathore

1. What is the output?

```
# include <stdio.h>
int main( )
{
int a[]={10,20,30,40,50,60};
int *p[]={a, a+1, a+2, a+3, a+4, a+5};
int **pp;
pp=p;
pp++;
printf("%d%d%d", pp-p, *pp-a, **pp);
*pp++;
printf("%d%d%d", pp-p, *pp-a, **pp);
++*pp;
printf("%d%d%d", pp-p, *pp-a, **pp);
++**pp;
printf("%d%d%d", pp-p, *pp-a, **pp);
return 0 ;
}
```

1 1 20
2 2 30
2 3 40
2 3 41



To access a[0] $\longrightarrow$ a[0] = * (a) = *p[0] = **(p+0) = **(pp+0) = 10

the pointer arithmetic is used with the double pointer.

An array of 6 elements is defined which is pointed by an array of pointer p.

The pointer array p is pointed by a double pointer pp. The elements of p are the pointers that are pointing to every element of the array a. Since we know that the array name contains the base address of the array hence, it will work as a pointer and can the value can be traversed by using *(a), *(a+1), etc. As shown in the image, a[0] can be accessed in the following ways.

- a[0]: it is the simplest way to access the first element of the array
- (a): since a store the address of the first element of the array, we can access its value by using indirection pointer on it.
- p[0]: if a[0] is to be accessed by using a pointer p to it, then we can use indirection operator (*) on the first element of the pointer array p, i.e., *p[0].
- *(pp): as pp stores the base address of the pointer array, *pp will give the value of the first element of the pointer array that is the address of the first element of the integer array. **p will give the actual value of the first element of the integer array.

Coming to the program, Line 1 and 2 declare the integer and pointer array relatively. Line 3 initializes the double pointer to the pointer array p. As shown in the image, if the address of the array starts from 200 and the size of the integer is 2, then the pointer array will contain the values as 200, 202, 204, 206, 208, 210. Let us consider that the base address of the pointer array is 300; the double pointer pp contains the address of pointer array, i.e., 300. Line number 4 increases the value of pp by 1, i.e., pp will now point to address 302.

Line number 5 contains an expression which prints three values, i.e., pp - p, *pp - a, **pp. Let's calculate them each one of them.

- pp = 302, p = 300 => pp-p = (302-300)/2 => pp-p = 1, i.e., 1 will be printed.
- pp = 302, *pp = 202, a = 200 => *pp - a = 202 - 200 = 2/2 = 1, i.e., 1 will be printed.
- pp = 302, *pp = 202, *(*pp) = 20, i.e., 20 will be printed.

Therefore as the result of line 5, The output 1, 1, 206 will be printed on the console. On line 6, *pp++ is written. Here, we must notice that two unary operators * and ++ will have the same precedence. Therefore, by the rule of associativity, it will be evaluated from right to left. Therefore the expression *pp++ can be rewritten as (*(pp++)). Since, pp = 302 which will now become, 304. *pp will give 204.

On line 7, again the expression is written which prints three values, i.e., pp-p, *pp-a, *pp. Let's calculate each one of them.

- pp = 304, p = 300 => pp - p = (304 - 300)/2 => pp-p = 2, i.e., 2 will be printed.
- pp = 304, *pp = 204, a = 200 => *pp-a = (204 - 200)/2 = 2, i.e., 2 will be printed.
- pp = 304, *pp = 204, *(*pp) = 30, i.e., 30 will be printed.

Therefore, as the result of line 7, The output 2, 2, 300 will be printed on the console. On line 8, ++*pp is written. According to the rule of associativity, this can be rewritten as, (++(*(pp))). Since, pp = 304, *pp = 204, the value of *pp = *(p[2]) = 206 which will now point to a[3].

On line 9, again the expression is written which prints three values, i.e., pp-p, *pp-a, *pp. Let's calculate each one of them.

- pp = 304, p = 300 => pp - p = (304 - 300)/2 => pp-p = 2, i.e., 2 will be printed.
- pp = 304, *pp = 206, a = 200 => *pp-a = (206 - 200)/2 = 3, i.e., 3 will be printed.
- pp = 304, *pp = 206, *(*pp) = 40, i.e., 40 will be printed.

Therefore, as the result of line 9, the output 2, 3, 409 will be printed on the console. On line 10, ++**pp is writen. according to the rule of associativity, this can be rewritten as, (++(*(*(pp)))). pp = 304, *pp = 206, **pp = 409, ++**pp => *pp = *pp + 1 = 410. In other words, a[3] = 410.

On line 11, again the expression is written which prints three values, i.e., pp-p, *pp-a, *pp. Let's calculate each one of them.

- pp = 304, p = 300 => pp - p = (304 - 300)/2 => pp-p = 2, i.e., 2 will be printed.
- pp = 304, *pp = 206, a = 200 => *pp-a = (206 - 200)/2 = 3, i.e., 3 will be printed.
- On line 8, **pp = 41.

Therefore as the result of line 9, the output 2, 3, 41 will be printed on the console.

At last, the output of the complete program will be given as:

2

```
#include<stdio.h>
#include<math.h>
int main()
{
printf("\n Result : %f" , ceil(1.44) );
printf("\n Result : %f" , ceil(1.66) );
printf("\n Result : %f" , floor(1.44) );
printf("\n Result : %f" , floor(1.66) );
return 0;
}
```

```
Result : 2.000000
Result : 2.000000
Result : 1.000000
Result : 1.000000
```

3

```
#include<stdio.h>
int main()
{
enum status { pass, fail, atkt};
enum status stud1, stud2, stud3;
stud1 = pass;
stud2 = atkt;
stud3 = fail;
printf("%d, %d, %d\n", stud1, stud2, stud3);
return 0;
}
```

0, 2, 1

inum status{pass,fail,atkt} ,value assign to staus is in the form of array so index starts from 0,1,2,......

Now when we access the value of staus of array that are pass, fail, atkt

then we assign these as like stud1=pass, stud2=atkt,stud3=fail

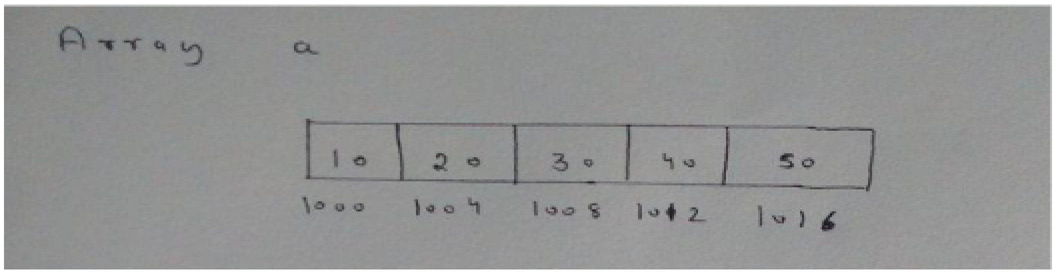
After executing this program output will be as

0,2,1

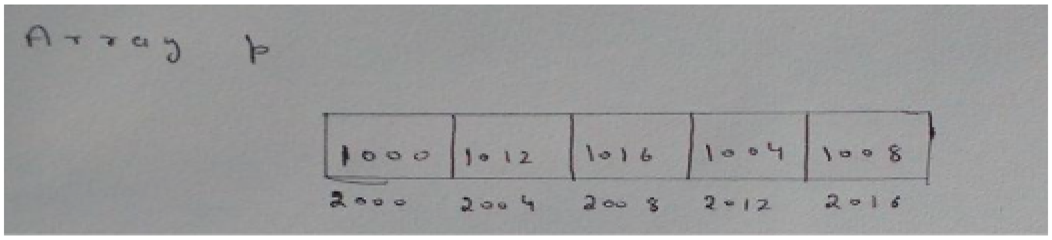
4

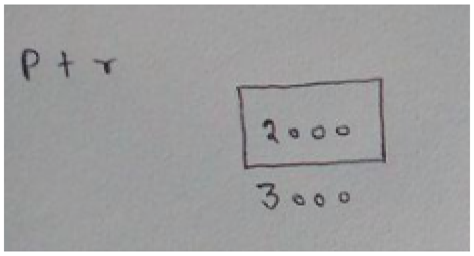
```
#include <stdio.h>
int
main ()
{
  static int a[] = { 10, 20, 30, 40, 50 };
  static int *p[] = { a, a + 3, a + 4, a + 1, a + 2 };
  int **ptr = p;
  ptr++;
printf ("%d %d", ptr - p, **ptr);
  }
```

1 40



Now ptr is actually pointing to the first element of array p. ptr++ will make it point to the next element of array p. Its value will then change to 2004**.**





One of the Rule of Pointer Arithmetic is that When you subtract two pointers, as long as they point into the same array, the result is the number of elements separating them

ptr is pointing to the second element and p is pointing to the first element so ptr-p will be equal to 1(Excluding the element to which ptr is pointing).

Now ptr = 2004 ——> *(2004) = 1012 —-> *(1012) —-> 40

Therefore, final answer is **140**.

5.

Assume base addr 2000

```
int main()
{
unsigned int x[4][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}, {10, 11, 12}};
printf("%u, %u, %u", x+3, *(x+3), *(x+2)+3); }
```

https://www.google.com/amp/s/www.geeksforgeeks.org/pointer-array-array-pointer/amp/

Address of x is 2000

X being 2 D array

x+3 = x+ 3*size

x +3 * 3*4

2000 + 36 = 2036.
+3) returns the value at address 2036. But since is 2 Darray, one* will just return the 1D array which is the starting address of it, which is 2036 only.
+2) = 2000 + 2 *3 * 4 = 2024

- +3 = 2024 + 3 *4 = 2036 (The changes the data type from 2D to 1D and hence +3 will add 3 4 and not 3 3 4)

```
The expression, *(x + 3) also prints same
address as x is 2D array.


The expression *(x + 2) + 3 = 2000 + 2*3*4 + 3*4
                            = 2036
```

Ans:-

2036, 2036, 2036

6

```
#include <stdio.h>
int fun(int arr[]) {
arr = arr+1;
printf("%d ", arr[0]);
}
int main(void) {
int arr[2] = {10, 20};
fun(arr);
printf("%d", arr[0]);
return 0;
}
```

20 10

7

```
#include <stdio.h>
int
main ()
{
  float arr[5] = { 12.5, 10.0, 13.5, 90.5, 0.5 };
  float *ptr1 = &arr[0];
  float *ptr2 = ptr1 + 3;
  printf ("%f ", *ptr2);
  printf ("%d", ptr2 - ptr1);
  return 0;
}
```

90.500000 3

Assume that the base address of the array 'arr' is 1000. So, the value 1000 gets stored in the pointer variable 'ptr1'. In the statement *ptr2 = ptr1 + 3;, the compiler actually does the operation of *ptr2 = ptr1 + (3*size of data type pointed by ptr1);. Hence, ptr2 becomes 1012. The value stored at the address 1012 is 90.5 and the same gets printed using the first printf function.

Pointer subtraction is possible only when both the pointers point to the variable of the same data type. When two pointer variables are subtracted, the compiler actually does the operation of (ptr2 - ptr1)/size of data type pointed by the pointer. ptr2 is 1012 and ptr1 is1000. (1012 - 1000)/4 = 3. Hence, the value 3 gets printed using the last printf function.

8.

```
#include <stdio.h>
int main()

{
int *ptr;
int x;
ptr = &x;
*ptr = 0;
printf(" x = %d\n", x);
printf(" *ptr = %d\n", *ptr);
*ptr += 5;
printf(" x = %d\n", x);
printf(" *ptr = %d\n", *ptr);
(*ptr)++;
printf(" x = %d\n", x);
printf(" *ptr = %d\n", *ptr);
return 0;
}
```

x = 0
*ptr = 0
x = 5
*ptr = 5
x = 6
*ptr = 6

9.

```
#include <stdio.h>
int
main ()
{
  int arri[] = { 1, 2, 3 };
  int *ptri = arri;
  char arrc[] = { 1, 2, 3 };
  char *ptrc = arrc;
  printf ("sizeof arri[] = %d ", sizeof (arri));
  printf ("sizeof ptri = %d ", sizeof (ptri));
  printf ("sizeof arrc[] = %d ", sizeof (arrc));
  printf ("sizeof ptrc = %d ", sizeof (ptrc));
  return 0;
}
```

sizeof arri[] = 12

sizeof ptri = 8

sizeof arrc[] = 3

sizeof ptrc = 8

10

```
#include<stdio.h>
int main( )
{
int arr[] = {10, 20, 30, 40, 50, 60};
int *ptr1 = arr;
int *ptr2 = arr + 5;
printf("Number of elements between two pointer are: %d.",
(ptr2 - ptr1));
printf("Number of bytes between two pointers are: %d",
(char*)ptr2 - (char*) ptr1);
return 0;
}
```

Number of elements between two pointer are: 5.Number of bytes between two pointers are: 20

11

```
int f(int n)
{
static int r = 0;
if (n <= 0) return 1;
if (n > 3)
{
r = n;
return f(n-2)+2;
}
return f(n-1)+r;
}
int main()
{
printf("%d", f(5));
}
```

18

f(5) = f(3)+2
The line "r = n" changes value of r to 5. Since r
is static, its value is shared for calls
calls. Also, all subsequent calls don't change r
because the statement "r = n" is in a if condition
with n > 3.
f(3) = f(2)+5
f(2) = f(1)+5
f(1) = f(0)+5
f(0) = 1

So f(5) = 1+5+5+5+2 = 18

12

```
int main()
{
static int i=5;
if(--i){
```

```
main();
printf("%d ",i);
}
}
```

0 0 0 0

13.

```
int main()
{
int x = 5;
int * const ptr = &x;
++(*ptr);
printf("%d", x);
return 0;
}
```

6

int *const is a constant pointer to integer This means that the variable being declared is a constant pointer pointing to an integer. Effectively, this implies that the pointer shouldn’t point to some other address. Const qualifier doesn’t affect the value of integer in this scenario so the value being stored in the address is allowed to change.

14

```
int
main ()
{
  register int i = 10;
  int *ptr = &i;
  printf ("%d", *ptr);
  return 0;
}
```

`error: address of register variable ‘i’ requested`

because when we say a variable is a register, it may be stored in a register instead of memory and accessing the address of a register is invalid

15

```
int main(void)
{
int i = 10;

const int *ptr = &i;
*ptr = 100;
printf("i = %d\n", i);
return 0;
}
```

`error: assignment of read-only location ‘*ptr’`

remove const

i=100

int const* is pointer to constant integer This means that the variable being declared is a pointer, pointing to a constant integer. Effectively, this implies that the pointer is pointing to a value that shouldn’t be changed. Const qualifier doesn’t affect the pointer in this scenario so the pointer is allowed to point to some other address.

16

```
#include <stdio.h>
```

```
int main()
{
int i = 1024;
for (; i; i >>= 1)
printf("RGIPT");
return 0;
}
```

1024 to binary 10000000000

rights shift means divide by 1

go on till I become 0

RGIPT RGIPT RGIPT RGIPT RGIPT RGIPT RGIPT RGIPT RGIPT RGIPT RGIPT