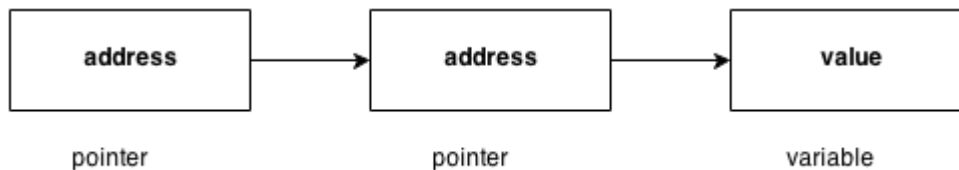


C Double Pointer (Pointer to Pointer)

As we know that, a pointer is used to store the address of a variable in C. Pointer reduces the access time of a variable. However, In C, we can also define a pointer to store the address of another pointer. Such pointer is known as a double pointer (pointer to pointer). The first pointer is used to store the address of a variable whereas the second pointer is used to store the address of the first pointer. Let's understand it by the diagram given below.



The syntax of declaring a double pointer is given below.

```
int **p; // pointer to a pointer which is pointing to an integer.
```

Example

```
#include <stdio.h>

int main () {

    int  var;
    int  *ptr;
    int  **pptr;

    var = 3000;

    /* take the address of var */
    ptr = &var;

    /* take the address of ptr using address of operator & */
    pptr = &ptr;

    /* take the value using pptr */
    printf("Value of var = %d\n", var );
    printf("Value available at *ptr = %d\n", *ptr );
    printf("Value available at **pptr = %d\n", **pptr);

    return 0;
}
```

OUTPUT

```
Value of var = 3000
Value available at *ptr = 3000
Value available at **pptr = 3000
```

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

```
void main ()
```

```
{
```

```
    int a = 10;
```

```
    int *p;
```

```
    int **pp;
```

```
    p = &a; // pointer p is pointing to the address of a
```

```
    pp = &p; // pointer pp is a double pointer pointing to the address of pointer p
```

```
    printf("address of a: %x\n",p); // Address of a will be printed
```

```
    printf("address of p: %x\n",pp); // Address of p will be printed
```

```
    printf("value stored at p: %d\n",*p); // value stored at the address contained by p i.e. 10 will be printed
```

```
    printf("value stored at pp: %d\n",**pp); // value stored at the address contained by the pointer stored at pp
```

```
}
```

OUTPUT

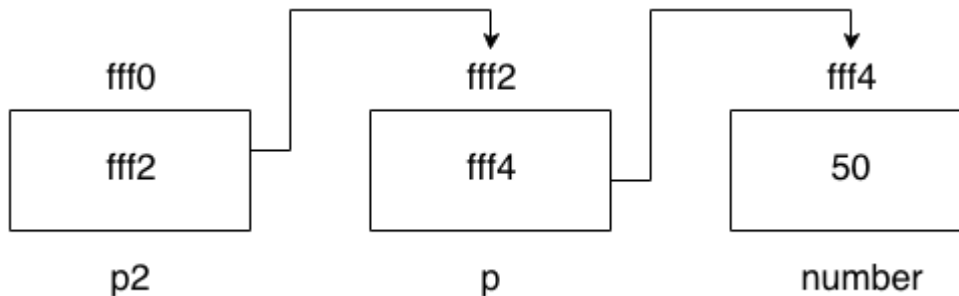
```
address of a: d26a8734
```

```
address of p: d26a8738
```

```
value stored at p: 10
```

```
value stored at pp: 10
```

C double pointer example



As you can see in the above figure, p2 contains the address of p (fff2), and p contains the address of number variable (fff4).

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

```
int main(){
```

```
    int number=50;
```

```
    int *p;//pointer to int
```

```
    int **p2;//pointer to pointer
```

```
    p=&number;//stores the address of number variable
```

```
    p2=&p;
```

```
    printf("Address of number variable is %x \n",&number);
```

```
    printf("Address of p variable is %x \n",p);
```

```
    printf("Value of *p variable is %d \n",*p);
```

```

printf("Address of p2 variable is %x \n",p2);
printf("Value of **p2 variable is %d \n",*p);
return 0;
}

```

OUTPUT

```

Address of number variable is fff4
Address of p variable is fff4
Value of *p variable is 50
Address of p2 variable is fff2
Value of **p variable is 50

```

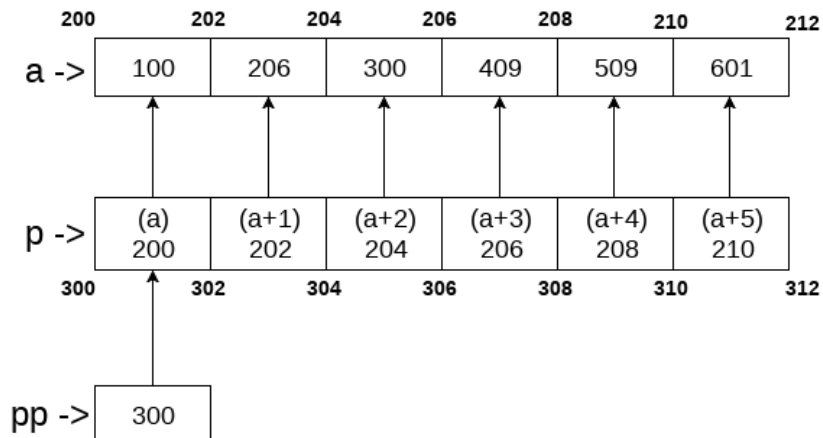
What will be the output of the following program?

```

1. #include<stdio.h>
2. void main ()
3. {
4.     int a[10] = { 100, 206, 300, 409, 509, 601 }; //Line 1
5.     int *p[] = { a, a+1, a+2, a+3, a+4, a+5 }; //Line 2
6.     int **pp = p; //Line 3
7.     pp++; // Line 4
8.     printf("%d %d %d\n",pp-p,*pp - a,**pp); // Line 5
9.     *pp++; // Line 6
10.    printf("%d %d %d\n",pp-p,*pp - a,**pp); // Line 7
11.    ++*pp; // Line 8
12.    printf("%d %d %d\n",pp-p,*pp - a,**pp); // Line 9
13.    ++**pp; // Line 10
14.    printf("%d %d %d\n",pp-p,*pp - a,**pp); // Line 11
15. }

```

Explanation



To access $a[0]$ $\rightarrow a[0] = * (a) = *p[0] = ** (p+0) = ** (pp+0) = 100$

In the above question, 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. However, the above image gives you a brief idea about how the memory is being allocated to the array a and the pointer array p. 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.

- $pp = 302, p = 300 \Rightarrow pp - p = (302 - 300) / 2 \Rightarrow pp - p = 1$, i.e., 1 will be printed.
- $pp = 302, *pp = 202, a = 200 \Rightarrow *pp - a = 202 - 200 = 2 / 2 = 1$, i.e., 1 will be printed.
- $pp = 302, *pp = 202, *(*pp) = 206$, i.e., 206 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.

- $pp = 304, p = 300 \Rightarrow pp - p = (304 - 300) / 2 \Rightarrow pp - p = 2$, i.e., 2 will be printed.
- $pp = 304, *pp = 204, a = 200 \Rightarrow *pp - a = (204 - 200) / 2 = 2$, i.e., 2 will be printed.
- $pp = 304, *pp = 204, *(*pp) = 300$, i.e., 300 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` \Rightarrow `pp - p = (304 - 300)/2` \Rightarrow `pp-p = 2`, i.e., 2 will be printed.
- `pp = 304`, `*pp = 206`, `a = 200` \Rightarrow `*pp-a = (206 - 200)/2 = 3`, i.e., 3 will be printed.
- `pp = 304`, `*pp = 206`, `*(pp) = 409`, i.e., 409 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 written. according to the rule of associativity, this can be rewritten as, `((++(**pp)))`. `pp = 304`, `*pp = 206`, `**pp = 409`, `++**pp` \Rightarrow `*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` \Rightarrow `pp - p = (304 - 300)/2` \Rightarrow `pp-p = 2`, i.e., 2 will be printed.
- `pp = 304`, `*pp = 206`, `a = 200` \Rightarrow `*pp-a = (206 - 200)/2 = 3`, i.e., 3 will be printed.
- On line 8, `**pp = 410`.

Therefore, as the result of line 9, the output 2, 3, 410 will be printed on the console.

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

Output

```
1 1 206
2 2 300
2 3 409
2 3 410
```

const Pointer in C

Constant Pointers

A constant pointer in C cannot change the address of the variable to which it is pointing, i.e., the address will remain constant. Therefore, we can say that if a constant pointer is pointing to some variable, then it cannot point to any other variable.

Syntax of Constant Pointer

`<type of pointer> *const <name of pointer>;`

Declaration of a constant pointer is given below:

```
int *const ptr;
```

Let's understand the constant pointer through an example.

```
1. #include <stdio.h>
2. int main()
3. {
4.     int a=1;
5.     int b=2;
6.     int *const ptr;
7.     ptr=&a;
8.     ptr=&b;
9.     printf("Value of ptr is :%d",*ptr);
10.    return 0;
11. }
```

In the above code:

- We declare two variables, i.e., a and b with values 1 and 2, respectively.
- We declare a constant pointer.
- First, we assign the address of variable 'a' to the pointer 'ptr'.
- Then, we assign the address of variable 'b' to the pointer 'ptr'.
- Lastly, we try to print the value of the variable pointed by the 'ptr'.

Output

Compilation failed due to following error(s).

```
main.c: In function 'main':
main.c:16:8: error: assignment of read-only variable 'ptr'
    ptr=&a;
    ^
main.c:17:8: error: assignment of read-only variable 'ptr'
    ptr=&b;
    ^
```

In the above output, we can observe that the above code produces the error "assignment of read-only variable 'ptr'". It means that the value of the variable 'ptr' which 'ptr' is holding cannot be changed. In the above code, we are changing the value of 'ptr' from &a to &b, which is not possible with constant pointers. Therefore, we can say that the constant pointer, which points to some variable, cannot point to another variable.

Pointer to Constant

A pointer to constant is a pointer through which the value of the variable that the pointer points cannot be changed. The address of these pointers can be changed, but the value of the variable that the pointer points cannot be changed.

Syntax of Pointer to Constant

```
const <type of pointer>* <name of pointer>
```

Declaration of a pointer to constant is given below:

```
const int* ptr;
```

Let's understand through an example.

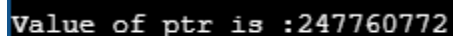
First, we write the code where we are changing the value of a pointer

```
1. #include <stdio.h>
2. int main()
3. {
4.     int a=100;
5.     int b=200;
6.     const int* ptr;
7.     ptr=&a;
8.     ptr=&b;
9.     printf("Value of ptr is :%u",ptr);
10.    return 0;
11. }
```

In the above code:

- We declare two variables, i.e., a and b with the values 100 and 200 respectively.
- We declare a pointer to constant.
- First, we assign the address of variable 'a' to the pointer 'ptr'.
- Then, we assign the address of variable 'b' to the pointer 'ptr'.
- Lastly, we try to print the value of 'ptr'.

Output



```
Value of ptr is :247760772
```

The above code runs successfully, and it shows the value of 'ptr' in the output.

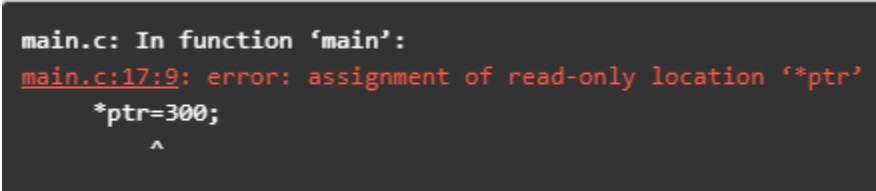
Now, we write the code in which we are changing the value of the variable to which the pointer points.

```
1. #include <stdio.h>
2. int main()
3. {
4.     int a=100;
5.     int b=200;
6.     const int* ptr;
7.     ptr=&b;
8.     *ptr=300;
9.     printf("Value of ptr is :%d",*ptr);
10.    return 0;
11. }
```

In the above code:

- We declare two variables, i.e., 'a' and 'b' with the values 100 and 200 respectively.
- We declare a pointer to constant.
- We assign the address of the variable 'b' to the pointer 'ptr'.
- Then, we try to modify the value of the variable 'b' through the pointer 'ptr'.
- Lastly, we try to print the value of the variable which is pointed by the pointer 'ptr'.

Output



```
main.c: In function 'main':
main.c:17:9: error: assignment of read-only location '*ptr'
    *ptr=300;
    ^
```

The above code shows the error "assignment of read-only location '*ptr'". This error means that we cannot change the value of the variable to which the pointer is pointing.

[Constant Pointer to a Constant](#)

A constant pointer to a constant is a pointer, which is a combination of the above two pointers. It can neither change the address of the variable to which it is pointing nor it can change the value placed at this address.

[Syntax](#)

const <type of pointer>* const <name of the pointer>;

Declaration for a constant pointer to a constant is given below:


```
const int* const ptr;
```

Let's understand through an example.

```
1. #include <stdio.h>
2. int main()
3. {
4.     int a=10;
5.     int b=90;
6.     const int* const ptr=&a;
7.     *ptr=12;
8.     ptr=&b;
9.     printf("Value of ptr is :%d",*ptr);
10.    return 0;
11. }
```

In the above code:

- We declare two variables, i.e., 'a' and 'b' with the values 10 and 90, respectively.
- We declare a constant pointer to a constant and then assign the address of 'a'.
- We try to change the value of the variable 'a' through the pointer 'ptr'.
- Then we try to assign the address of variable 'b' to the pointer 'ptr'.
- Lastly, we print the value of the variable, which is pointed by the pointer 'ptr'.

Output

Compilation failed due to following error(s).

```
main.c: In function 'main':
main.c:17:9: error: assignment of read-only location '*ptr'
    *ptr=12;
    ^
main.c:18:6: error: assignment of read-only variable 'ptr'
    ptr=&b;
    ^
```

The above code shows the error "assignment of read-only location '*ptr'" and "assignment of read-only variable 'ptr'". Therefore, we conclude that the constant pointer to a constant can change neither address nor value, which is pointing by this pointer.