

Pointers

Pointers

Fundamentals

Definition: A **pointer** is a variable that represents the **location** (rather than the **value**) of a data item, such as a variable or an array element.

Useful applications:

- Allows to **return multiple data items** from a function via function arguments.
- Allows **passing one function** to another function
- Allows **alternate ways to access arrays**
- Allows to **access group of strings** with ease
- Allows **passing structures** from one function to another function

Fundamentals...

How to access the address of a data item?

We need to know the address of the variable which stores this data item.

Address of operator (&)

*ff → logical and
binary*

Ex: If the variable name is **var** then the address can be accessed using **&var**

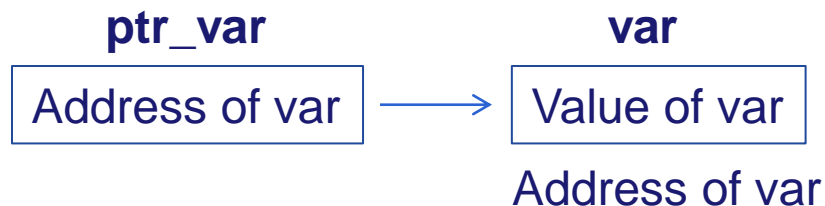
Unary operator address of

We can assign the address of V to another variable.

Ex: type `*ptr_var = &var`

This new variable `ptr_var` is called as a pointer variable which points to **var**.

PV points to the location where V is stored.



Fundamentals...

How to access the data item stored in **V** using **PV**?

Using **Indirection Operator (*)**.

Ex: `*ptr_var`

Here, `*ptr_var` and `var` represents the same data item.

This new variable `ptr_var` is called as a pointer variable which points to `var`.

`ptr_var` points to the location where **V** is stored.



Pointers...

- Creation of a pointer

- `type *ptr_variable` - Creates a pointer variable

*int *ptr_variable;*

- Dereferencing a pointer

- `*ptr_variable` - Returns contents stored at address

- Indirect assignment

- `*ptr_variable = val` - Stores value at address pointed by the pointer

**p = m;*

- Assignment

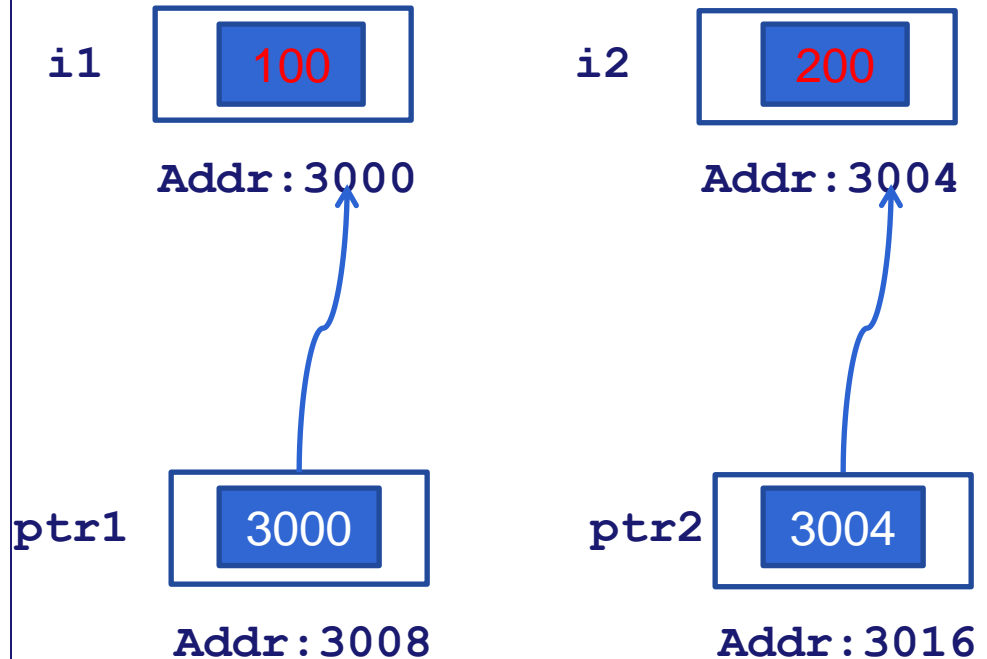
- `pointer = ptr` - Stores pointer in another variable

p = q

Using Pointers

How to use the pointers – A simple Example

```
int i1;  
int i2;  
int *ptr1;  
int *ptr2;  
  
i1 = 100;  
i2 = 200;  
  
ptr1 = &i1;  
ptr2 = &i2;
```



i1 value 100,
*ptr1 value 100

i2 value 200
*ptr2 value 200

Note: *pointer - Returns contents stored at address

Pointers

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

```
void main()
```

```
{
```

```
    int i1;
```

```
    int i2;
```

```
    int *ptr1;
```

```
    int *ptr2;
```

```
    i1 = 100;
```

```
    i2 = 200;
```

```
    ptr1 = &i1;
```

```
    ptr2 = &i2;
```

```
    printf("i1 address.....%u\n",&i1);
```

```
    printf("i2 address.....%u\n",&i2);
```

```
    printf("ptr1 address...%u\n",ptr1);
```

```
    printf("ptr2 address...%u\n",ptr2);
```

```
    printf("ptr1 address...%u\n",&ptr1);
```

```
    printf("ptr2 address...%u\n",&ptr2);
```

```
        printf("*ptr1 ...%d\n",*ptr1);
```

```
        printf("*ptr2 ...%d\n",*ptr2);
```

```
}
```


What is the Value of y?

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

```
int main()
```

```
{
```

```
    int x,y;
```

```
    int *ptr_p;
```

```
    x = 5;
```

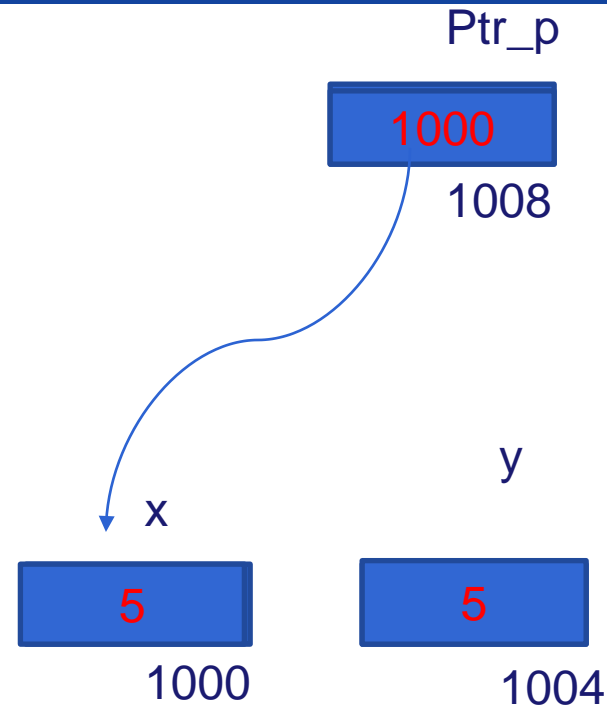
```
    ptr_p = &x;
```

```
    y = *ptr_p; // *ptr_p Returns contents stored at address
```

```
    printf("%d\n", y);
```

```
    return 0;
```

```
}
```



What is the Value of x?

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

```
int main()
```

```
{
```

```
    int x;
```

```
    int *ptr_p;
```

```
    x = 5;
```

```
    ptr_p = &x;
```

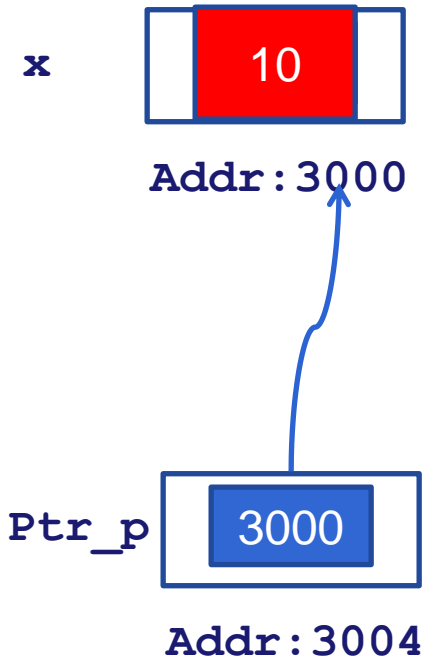
```
    *ptr_p = 10; // x changed to 10
```

```
    printf("%d\n", x);
```

```
    return 0;
```

```
}
```

Note: *ptr_p = 10 means , contents stored at address ptr_p is changed to 10



What is the Value ?

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

```
int main(void)
```

```
{
```

```
    int *ptr_p; // ptr-p contains garbage value
```

```
    printf("%d\n",*ptr_p); //run time error, ptr-p point nowhere
```

```
    return 0;
```

```
}
```

The result of this program is a segmentation fault, some other run-time error or the random address is printed. The meaning of a segmentation fault is that you have used a pointer that points to an invalid address.

In most cases, a pointer that is not initialized or a wrong pointer address is the cause of segmentation faults.

What is the Value of the a and b ?

```
void swapping(int c, int d)
```

```
{
```

```
    int tmp;
```

```
    tmp = c;
```

```
    c = d;
```

```
    d = tmp;
```

```
    printf("In function: %d %d\n", c , d);
```

```
}
```

```
void main( )
```

```
{
```

```
    int a,b;
```

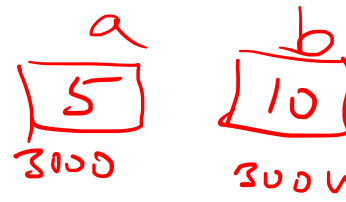
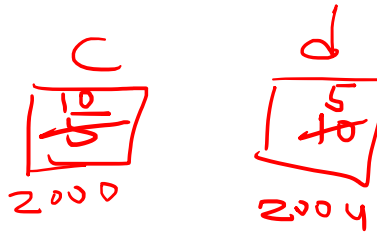
```
    a=5; b=10;
```

```
    printf("input: %d %d\n", a, b);
```

```
    swapping(a,b);
```

```
    printf("output: %d %d\n", a, b);
```

```
}
```



What is the Value of the a and b ?

```
void swapping(int *ptr_c, int *ptr_d) {
```

```
    int tmp;
```

```
    tmp = *ptr_c;
```

```
    *ptr_c = *ptr_d;
```

```
    *ptr_d = tmp;
```

```
}
```

```
void main()
```

```
{
```

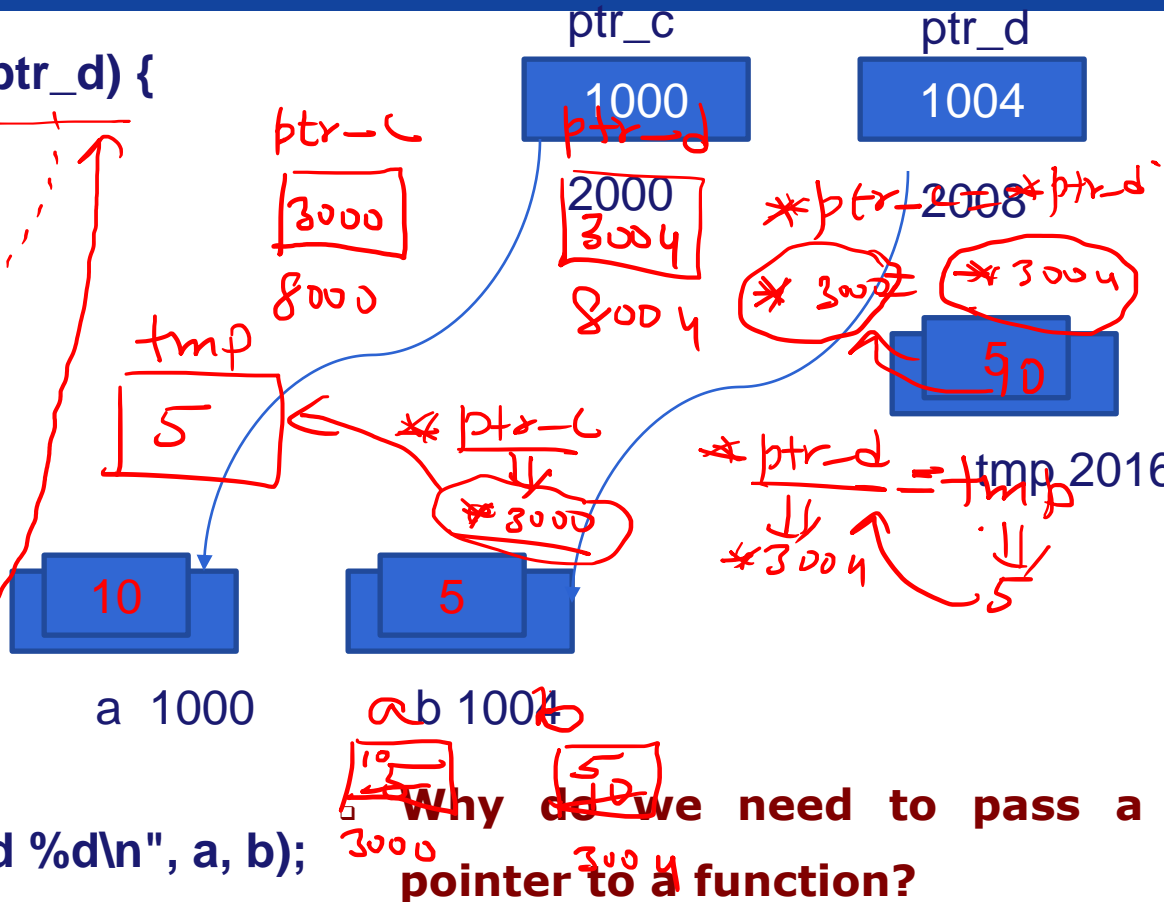
```
    int a=5,b=10;
```

```
    printf("Before swapping: %d %d\n", a, b);
```

```
    swapping(&a,&b);
```

```
    printf(" Before swapping : %d %d\n", a, b);
```

```
}
```



Why do we need to pass a pointer to a function?

- It allows a function to **alter or change** the value of a variable **outside their scope**.

Pointing to the same address

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

```
void main()
```

```
{
```

```
    int x;
```

```
    int *ptr_b , *ptr_c, *ptr_d;
```

```
    x=5;
```

```
    ptr_b = &x;
```

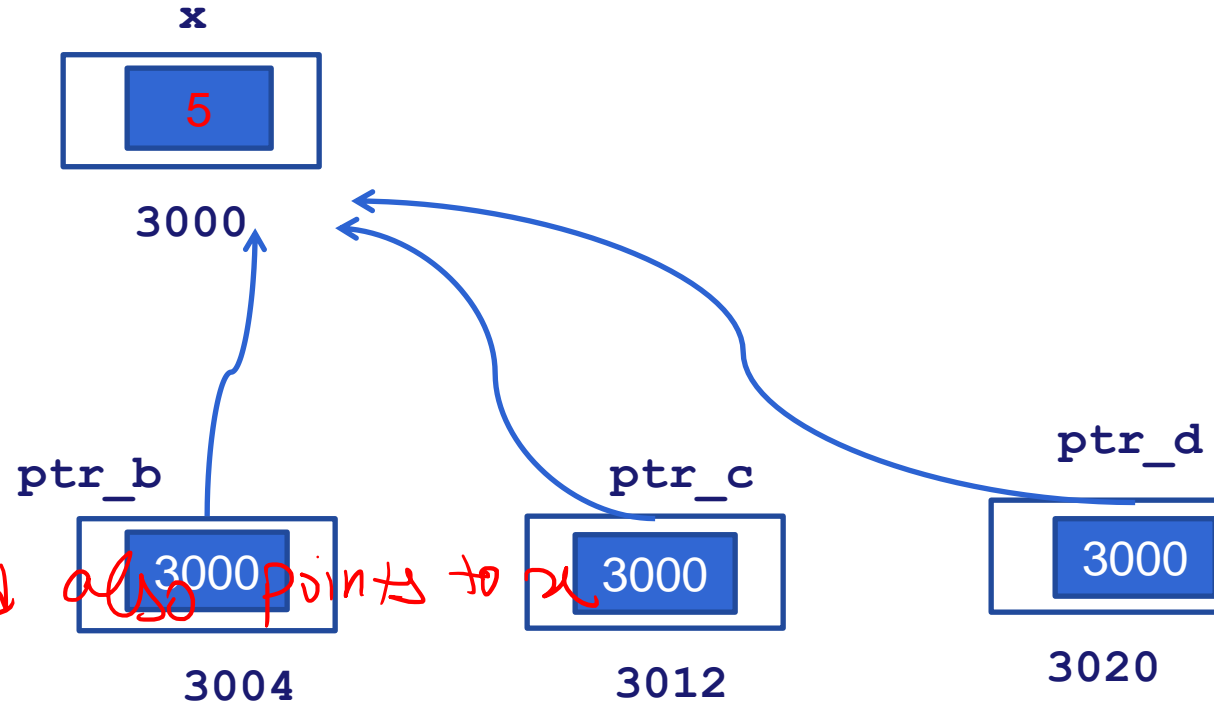
```
    ptr_c = &x;
```

```
    ptr_d = ptr_b; // ptr_d also points to x
```

```
    printf (" %d , %d , %d, %d ", x, *ptr_b ,
```

```
    *ptr_c, *ptr_d);
```

```
}
```



Using Pointers...

```
int  m      = 20;    /* Simple Integers */
int  n      = 30;

int *ptr1 = &m;    /* get addresses of data */
int *ptr2 = &n;

*ptr1 = ptr2;
```

What happens?

Type check warning: `ptr2` is not an `int`



Size of pointers

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

```
void main() {
```

```
    int a=10;
```

```
    char ch='A';
```

```
    int *ptr_b;
```

```
    char *ptr_c;
```

```
    ptr_b = &a;
```

```
    ptr_c = &ch;
```

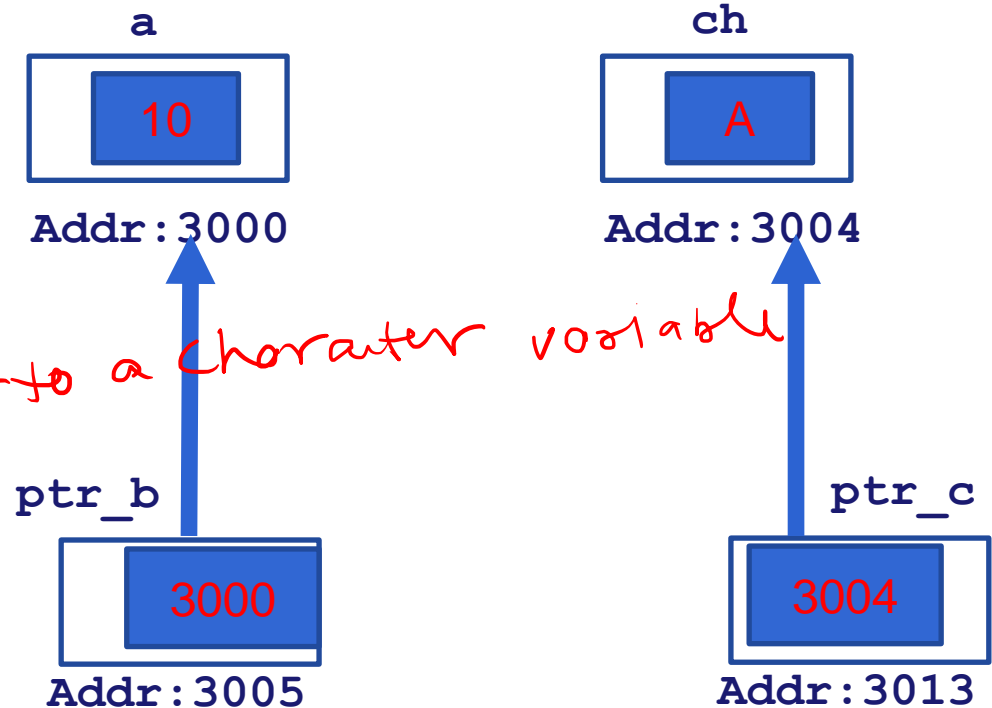
```
    printf (" %d \n",sizeof(a) );
```

```
    printf (" %d \n",sizeof(ch) );
```

```
    printf (" %d , %d , %d \n",a,*ptr_b,sizeof(ptr_b) );
```

```
    printf (" %c , %c , %d \n",ch,*ptr_c,sizeof(ptr_c) );
```

```
}
```



Quiz – pointer ex:

```
/* A program which shows a simple demo on pointer variables */
#include<stdio.h>

int main( )
{
    int U = 3, v;
    int *pu; /* pointer to an integer */
    int *pv; /* pointer to an integer */
    pu = &U; /* assign address of U to pu */
    v = *pu; /* assign value of U to v */
    pv = &v; /* assign address of v to pv */
    printf("\nu=%d &u=%p pu=%p *pu=%d", U, &U, pu, *pu);
    printf("\n\nv=%d &v=%p pv=%p *pv=%d", v, &v, pv, *pv);
    return 0;
}
```

sizeof pointers and arrays

```
int main()
{
    int a[10], i;
    int *ptr;
    ptr=&i;
    printf("%d\n", sizeof(ptr));
    printf("%d\n", sizeof(int *));
    printf("%d\n", sizeof(char *));
    printf("%d\n", sizeof(float *));
    printf("%d\n", sizeof(void *));
    printf("%d\n", sizeof(a));
}
```

Pointer Arithmetic

Addition: pointer + number

Subtraction: pointer - number

Example:

```
char *ptr;  
char a;  
char b;  
  
ptr = &a;  
ptr += 1;
```

Adds 1*sizeof(char) to
the memory address

```
int *p;  
int a;  
int b;  
  
p = &a;  
p += 1;
```

Adds 1*sizeof(int) to
the memory address

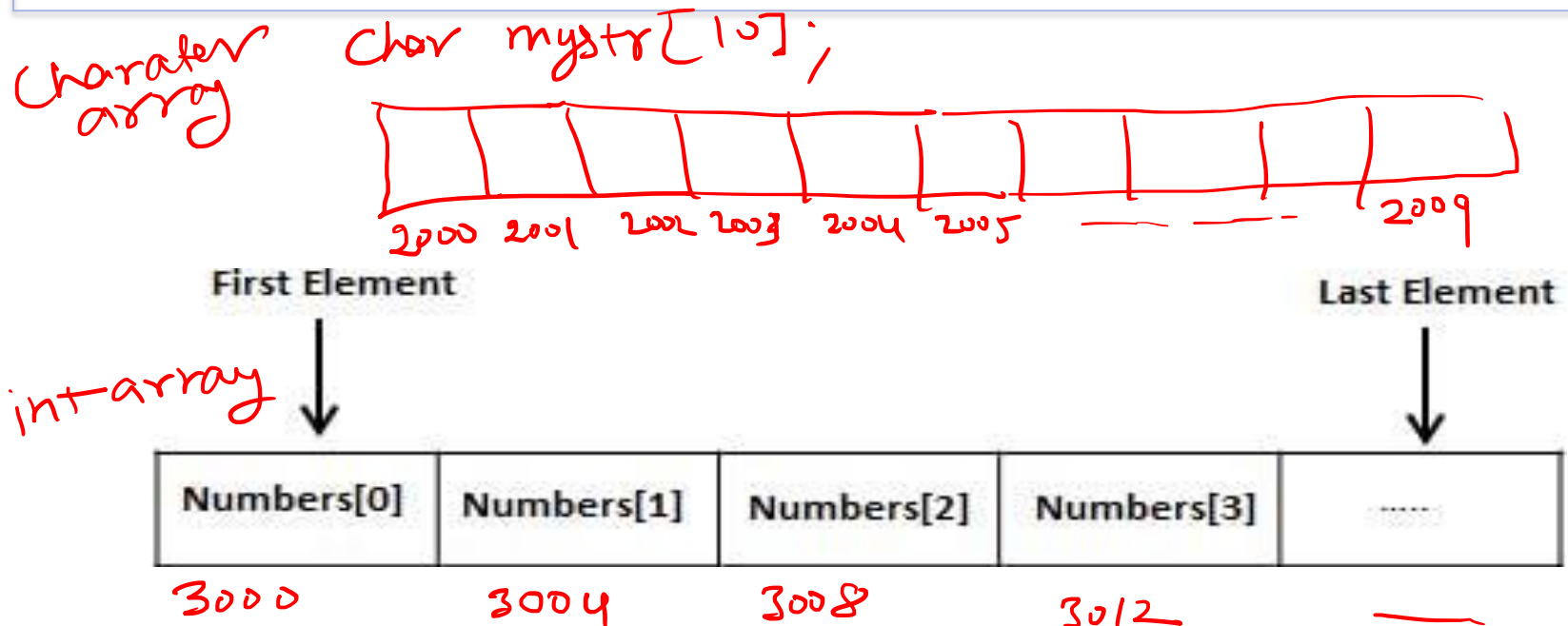
**In both the cases, p now points to b
(Assuming compiler doesn't reorder variables in memory)**

Warning: Pointer arithmetic need to be performed carefully

Arrays & Pointers

Review of Arrays

- **Definition:** Array is a derived data type. Array is a collections of variables of same type.
- Fixed size, size defines the number of elements in the array
- Stored in contiguous memory locations
- Array element is accessed by its index value



program on 1-D Array

```
#include <stdio.h>

int main()
{
    int num[] = {21,18,57,45,50};
    int i;
    for(i=0;i<5;i++)
    {
        printf("\n Integer Array Element num[%d] : %d",i,num[i]);
    }
    return 0;
}
```

Pointer Arithmetic

Example: `int A[10];`

Memory Allocation:

Base Address is :address of `A[0]` i.e `A`

If Base address is 1000

`sizeof(int) = 4`

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
Base Address	+0	+4	+8	+12	+16	etc.				

`A+1` means:

Adds $1 * \text{sizeof}(\text{int})$ to the memory address

`A+4` means:

Adds $4 * \text{sizeof}(\text{int})$ to the memory address

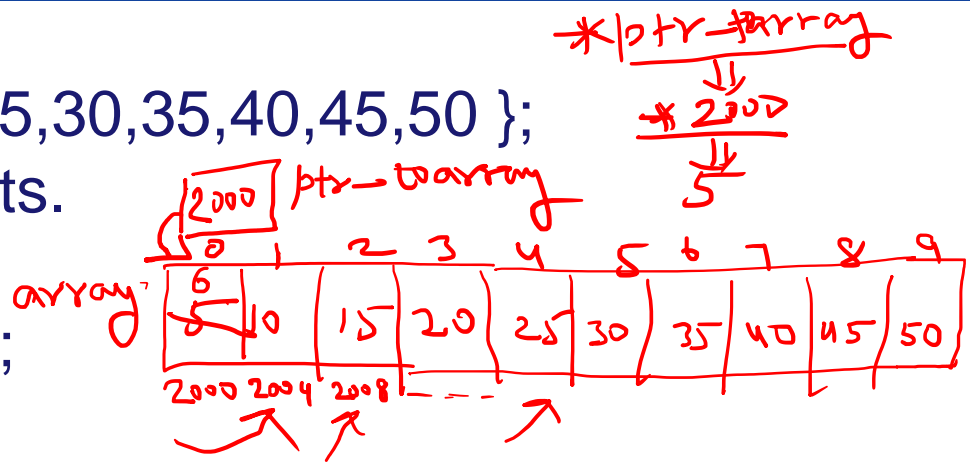
$\circ A[i] \equiv *(A+i)$

Warning: Pointer arithmetic need to be performed carefully

Pointers and arrays

```
int array[10]={5,10,15,20,25,30,35,40,45,50 };
// an array with ten elements.
```

```
int *ptr_toarray = &array[0];
```



*ptr_toarray is nothing but a 0th element in the array i.e array[0]

```
*(ptr_toarray + 2); // is same as array[2]
```

*(ptr_toarray + 2); // is same as array[2]

ptr_toarray + 2 ⇒ points to 2nd index element of array

*(ptr_toarray + 2) ⇒ 15

*(ptr_toarray + 2); is something different then *(ptr_toarray) + 2;

*++ptr_toarray; // 10 *ptr_toarray + 2 ⇒ 7

array+0 .. 0th element address

array+1 .. 1st element address

array+2 .. 2nd element address

*(array+0) .. 0th element

*(array+1) .. 1st element

*(array+2) .. 2nd element

Pointer incrementation

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

```
int main()
```

```
{
```

```
    int num[5] = {21,18,57,45,50};
```

```
    int i;
```

```
    int *iptr=num; // Can also write int *ptr = &num[0];
```

```
    for(i=0;i<5;i++)
```

```
    {
```

```
        printf("\n Integer Array Element num[%d] : %d  addr: %p\n",i,*iptr,iptr);
```

```
        iptr++; // go to next int location in num
```

```
    }
```

```
    return 0;
```

```
}
```

Pointer Decrement

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

```
int main()
```

```
{
```

```
    int num[5] = {21,18,57,45,50};
```

```
    int i;
```

```
    int *iptr=num+4; // or int *iptr = &num[4]
```

```
    for(i=0;i<5;i++)
```

```
    {
```

```
        printf("\n Integer Array Element num[%d] : %d  addr: %p\n",i,*iptr,iptr);
```

```
        iptr--;
```

```
    }
```

```
    return 0;
```

```
}
```

Arrays and Pointers

Passing arrays

- Array \approx pointer to the first (0th) array element
 - `a[i] \equiv *(a+i)`
- An array is passed to a function as a pointer
 - **The array size is lost !**
- Usually bad style to interchange arrays and pointers

`int *array` Must explicitly pass the size

```
int
foo(int array[],
    unsigned int size)
{
    ... array[size - 1] ...
}

int
main(void)
{
    int a[10], b[5];
    ... foo(a, 10) ... foo(b, 5) ...
}
```

Arrays and Pointers...

```
int
foo(int array[],
    unsigned int size)
{
    ...
    printf("%d\n", sizeof(array));
}

int
main(void)
{
    int a[10], b[5];
    ... foo(a, 10) ... foo(b, 5) ...
    printf("%d\n", sizeof(a));
}
```

What does this print?

Prints the size of an pointer: 8

(Because **array** is really
a pointer)

What does this print?

40