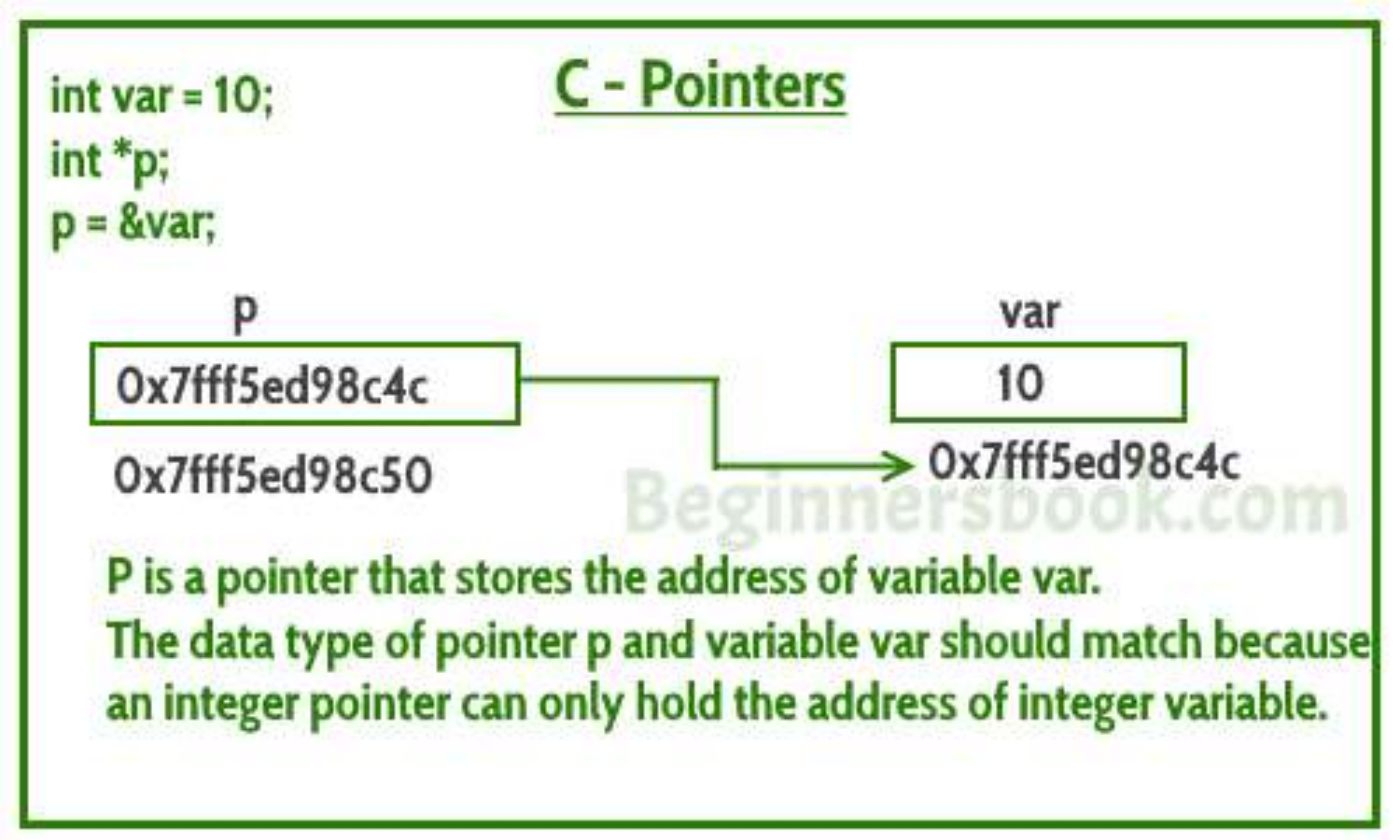
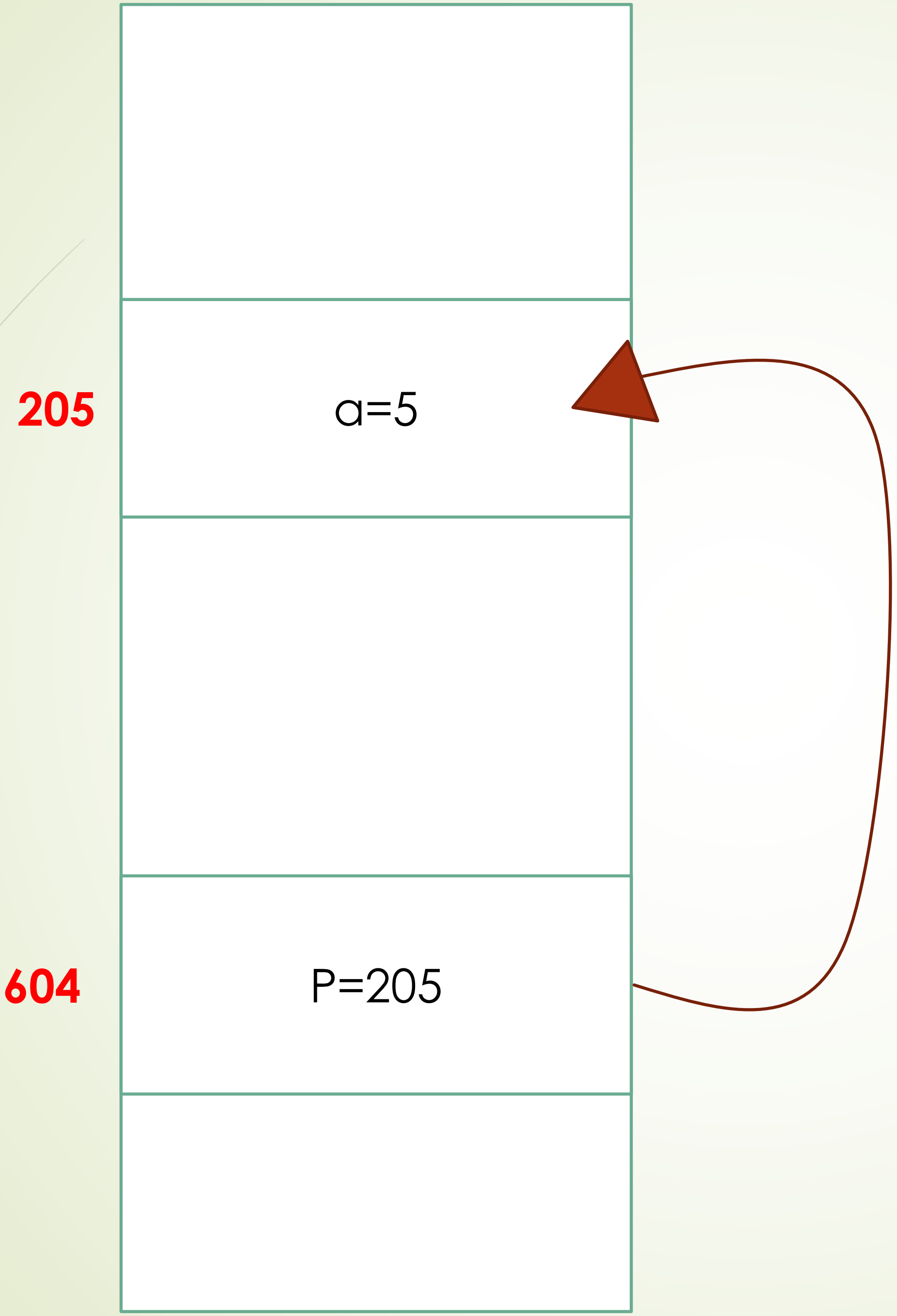
Pointers

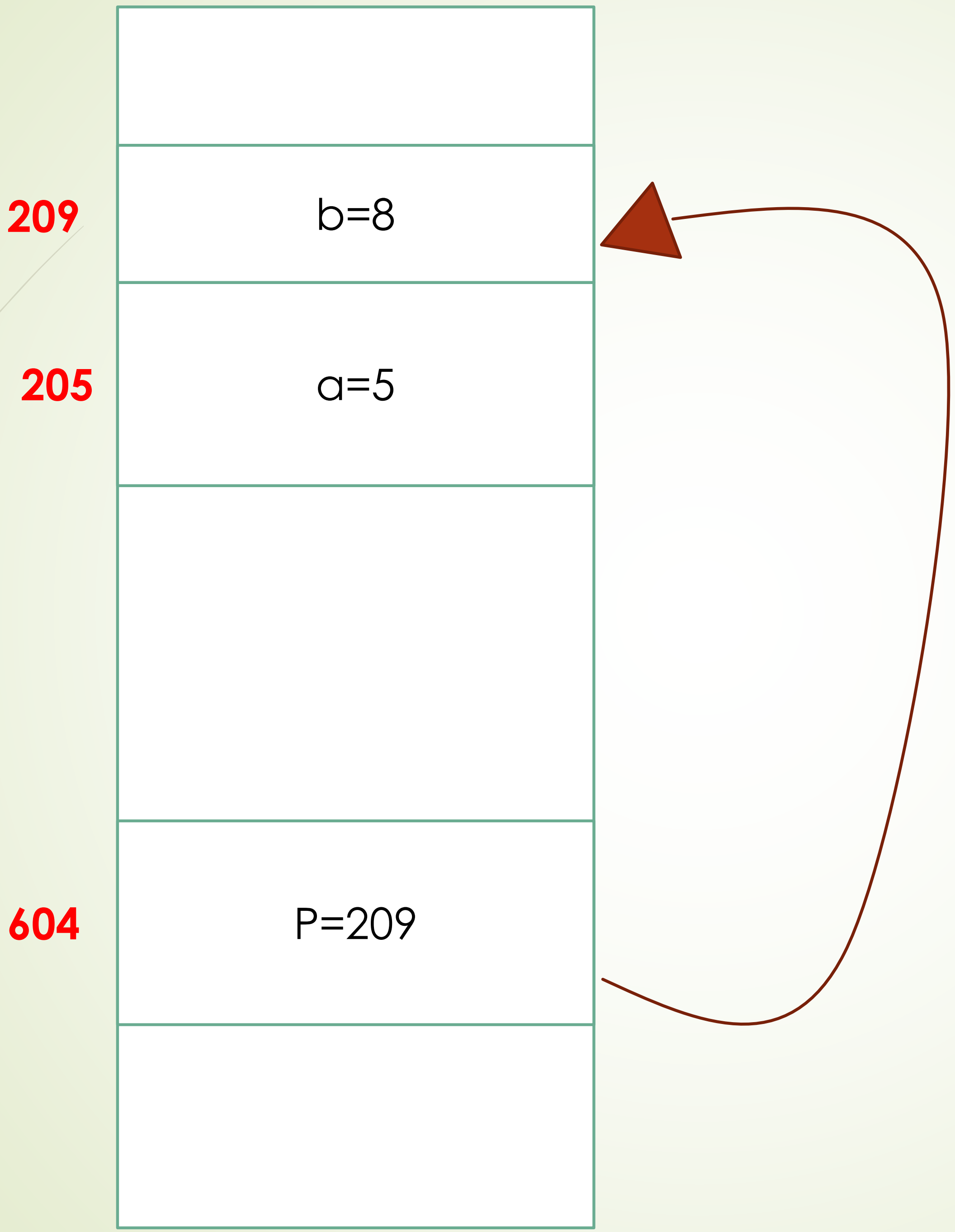
- Pointers are variables that store the address of other variables.
- The pointer variable might be belonging to any of the data type such as int, float, char, double, short etc.
- Normal variable stores the value whereas pointer variable stores the address of the variable.
- If you have a variable **var** in your program, **&var** will give you its address in the memory.
- Pointer Syntax : data_type *var_name; Example : int *p; char *p;
- Where, * is used to denote that "p" is pointer variable and not a normal variable.


C - Pointers
int var = 10;
int *p;
p = &var;
p
0x7fff5ed98c4c
0x7fff5ed98c50
var
10
0x7fff5ed98c4c
Beginnersbook.com
P is a pointer that stores the address of variable var.
The data type of pointer p and variable var should match because an integer pointer can only hold the address of integer variable.

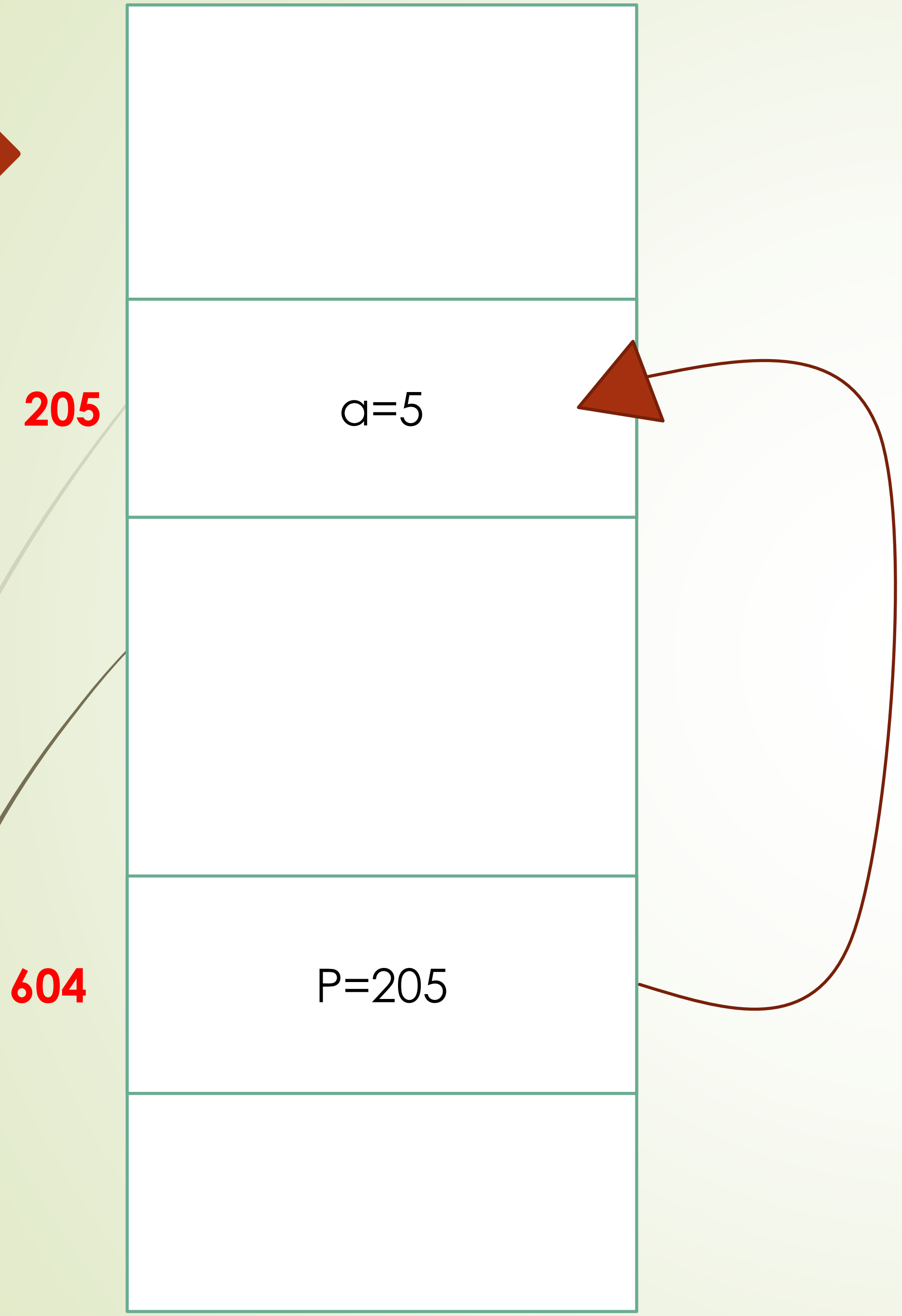


Int a;
Int *p;

P= &a;

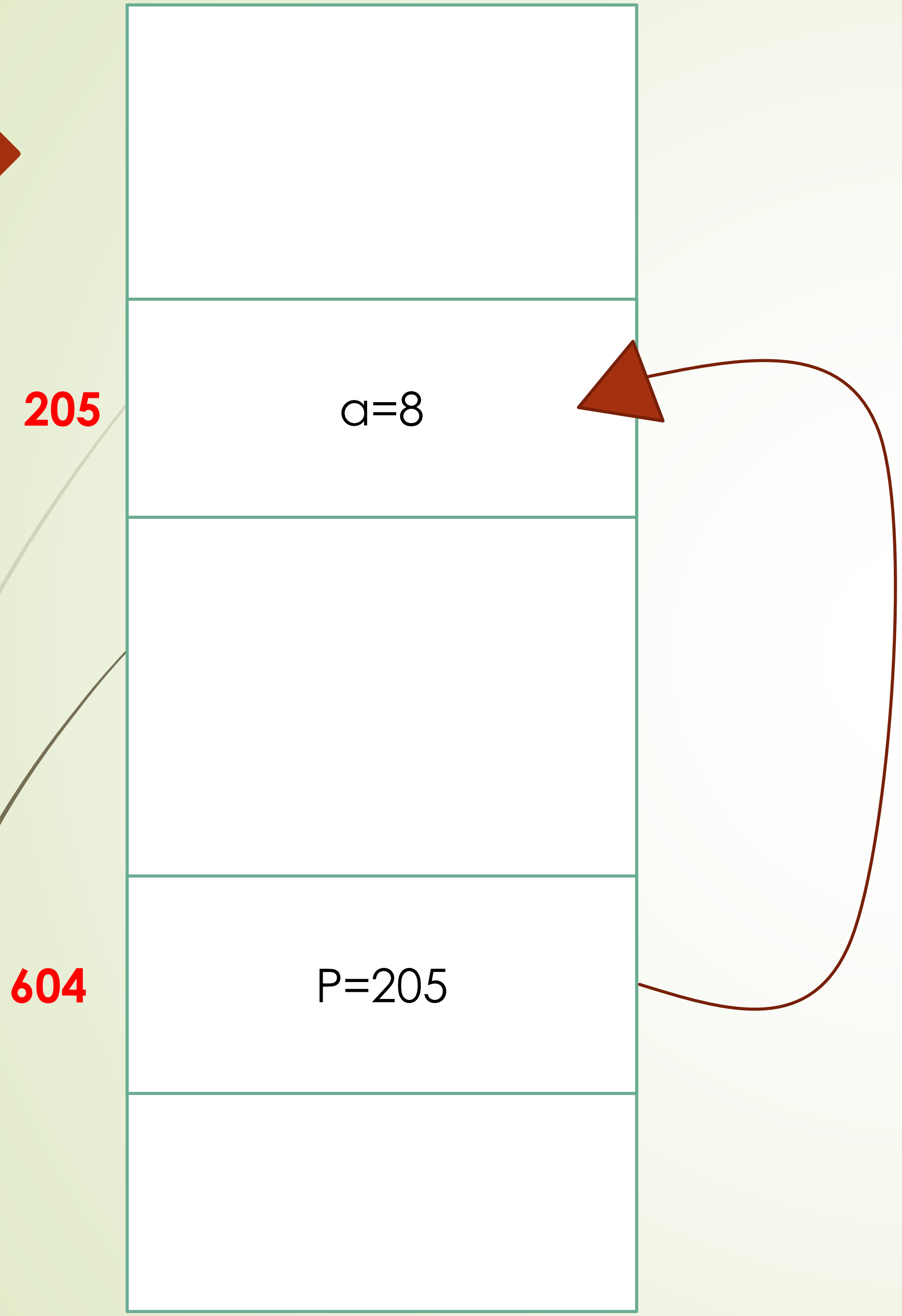



```
Int a;
Int *p;
Int b;
P=&b;
```



Assigning addresses to Pointers

- Int a=5;
- Int *p;
- we give the address to pointer as
  p=&a;
  Here, 5 is assigned to the a variable. And, the address of a is assigned to the p pointer.
- If we want to print p
  cout<<p; //205
  cout<<&a; //205
  cout<<&p; //604
  cout<<*p; //5   Derefrencing



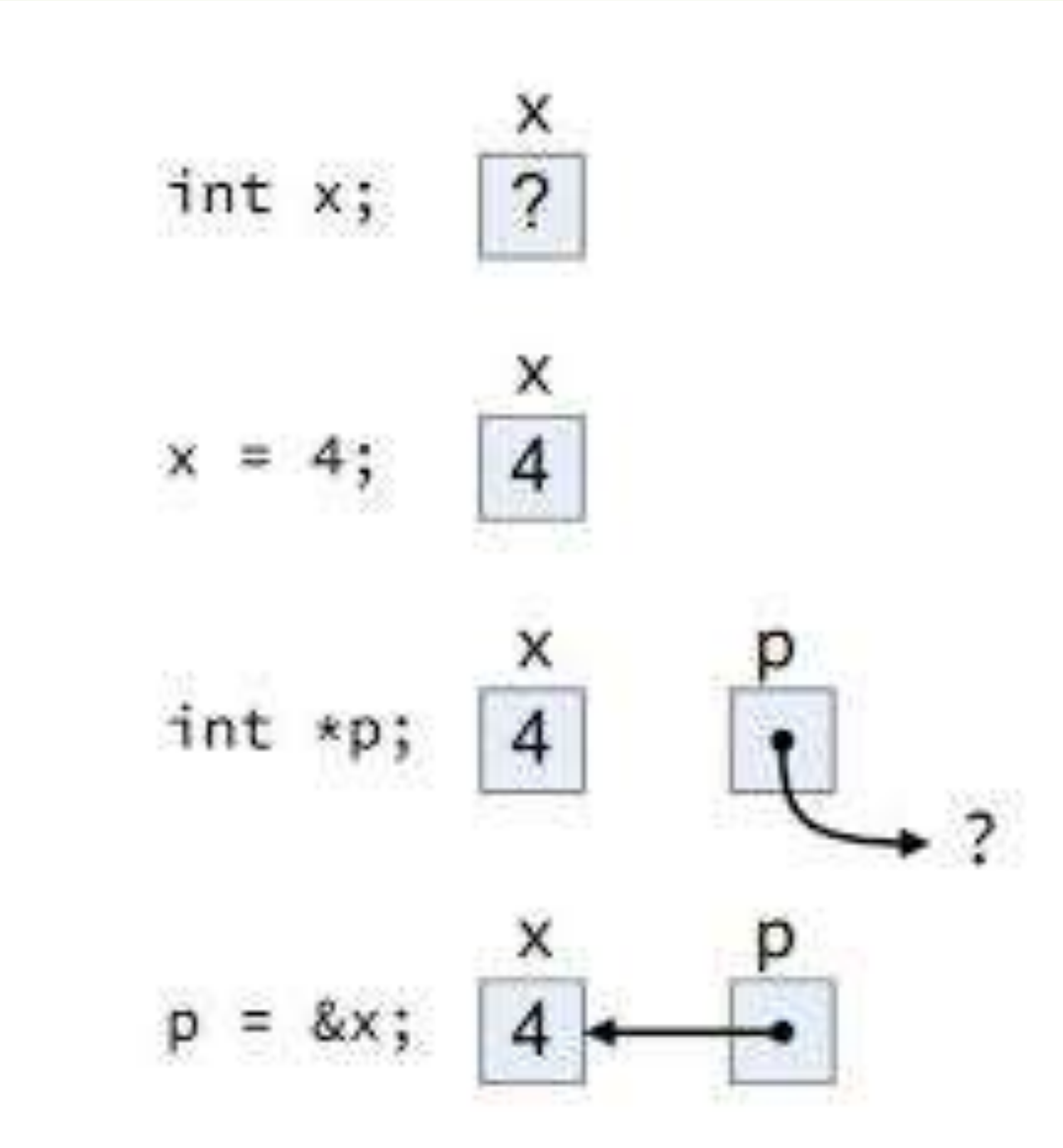
Assigning addresses to Pointers

- Int a=5;
- Int *p;
- we give the address to pointer as
  p=&a;
  Here, 5 is assigned to the a variable. And, the address of a is assigned to the p pointer.
- If we want to print p
  cout<<p; //205
  cout<<&a; //205
  cout<<&p; //604
  cout<<*p; //5   Derefrencing
  *p=8

  cout<<*p; //8

Some Important points

- Always C pointer is initialized to null, i.e. int *p = null.
- The value of null pointer is 0.
- & symbol is used to get the address of the variable.
- * symbol is used to get the value of the variable that the pointer is pointing to.
- If a pointer in C is assigned to NULL, it means it is pointing to nothing.
- Pointer addition, multiplication, division are not allowed. You have to deference it first then you can Add, divide and multiply etc.
  - *int   *ip;   /* pointer to an integer */*
  - *double *dp;   /* pointer to a double */*
  - *float *fp;   /* pointer to a float */*
  - *char   *ch   /* pointer to a character */*
- The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.


int x;
x
?
x = 4;
x
4
int *p;
x
4
p
?
p = &x;
x
4
p

Representation of integer

Suppose we take an integer
int a=1025;
1025 is representing as 32 bits because integer is of 4 bytes and every bytes contains 8 bits. So 1025 is representing the following way.

00000000	**00000000**	**00000100**	**00000001**
204	203	202	201

Void pointer

Void *p0;

P0=p1;

We can’t dereference it like we do in printing , and we can’t do this p0+1
We will discuss about this later.

Common mistakes

- int c, *pc;
- Pc=&c;
- // pc is address but c is not
- pc = c; // Error

- // &c is address but *pc is not
- *pc = &c; // Error
-

- // both &c and pc are addresses
- pc = &c;

- // both c and *pc values
- *pc = c;

Do not confuse on it

```
int main()
{
    int c = 5;
    int *p = &c;
    cout<<*p; // 5
    return 0;
}
```

Why didn't we get an error when using
int *p = &c;?

It's because

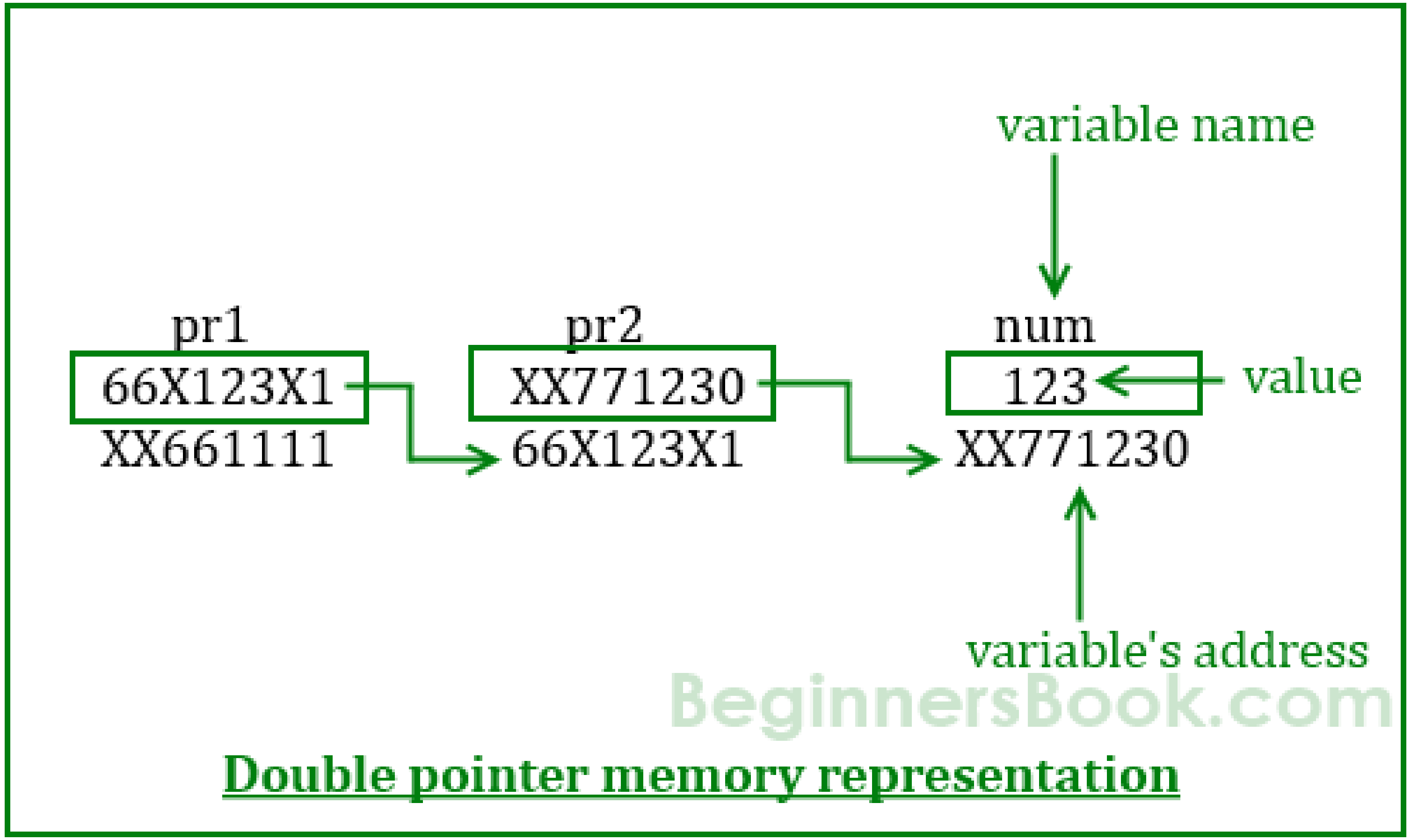
int *p = &c;
is equivalent to

int *p:
p = &c;

Find Sum
Find MAx

Pointers to Pointers

- When a pointer holds the address of another pointer then such type of pointer is known as **pointer-to-pointer** or **double pointer**.
- int **pr1;
  pr1=&pr2;



Pointers and arrays

- Suppose we declare an array arr,
- int arr[5] = { 1, 2, 3, 4, 5 };
- Assuming that the base address (i.e address of the first element of the array) of arr is 1000 and each integer requires two bytes, the five elements will be stored as follows:

element	arr[0]	arr[1]	arr[2]	arr[3]	arr[4]
Address	1000	1002	1004	1006	1008

- Here variable arr will give the base address, which is a constant pointer pointing to the first element of the array, arr[0]. Hence arr contains the address of arr[0] i.e 1000. In short, arr has two purpose - it is the name of the array and it acts as a pointer pointing towards the first element in the array.
- arr is equal to &arr[0] by default
- We can also declare a pointer of type int to point to the array arr.

```
int *p;

p = arr;

// or,

p = &arr[0];    //both the statements are equivalent.
```

Similarly if you print *(arr+1) it is same as arr[1]

Now we can access every element of the array arr using p++ to move from one element to another.

```
int main()
{
    int i;
    int a[5] = {1, 2, 3, 4, 5};
    int *p = a;     // same as int*p = &a[0]
    for (i = 0; i < 5; i++)
    {
        cout<<*p;
        p++;
    }

    return 0;
}
```

In the above program, the pointer *p will print all the values stored in the array one by one. We can also use the Base address (a in above case) to act as a pointer and print all the values.

Array to Function as parameters

- Arrays always passed as referenced parameter.
- It never get copied completely.
- We cant increment and decrement like pointer

Why we use Pointers

- **Pointers** are used to store and manage the addresses of dynamically allocated blocks of memory. Such blocks are used to store data objects or arrays of objects.
- Most structured and object-oriented languages provide an area of memory, called the heap or free store, from which objects are dynamically allocated.
- you can even use **++** and **--** with a pointer, but not with an array name because this is a constant pointer and cannot be changed. So to summarise: An array's name is a ***constant pointer*** to the first element in the array that is **a==&a[0]** and ***a==a[0]**.

Array to Function as parameters

```c
#include<stdio.h>
int SumOfElements(int* A, int size)// "int* A" or "int A[]"  ..it's the same..
{
    int i, sum = 0;

    for(i = 0;i< size;i++)
    {
        sum+= A[i]; // A[i] is *(A+i)
    }
    return sum;
}
int main()
{
    int A[]= {1,2,3,4,5};
    int size = sizeof(A)/sizeof(A[0]);
    int total = SumOfElements(A,size); // A can be used for &A[0]
    printf("Sum of elements = %d\n",total);
    printf("Main - Size of A = %d, size of A[0] = %d\n",sizeof(A),sizeof(A[0]));
}
```

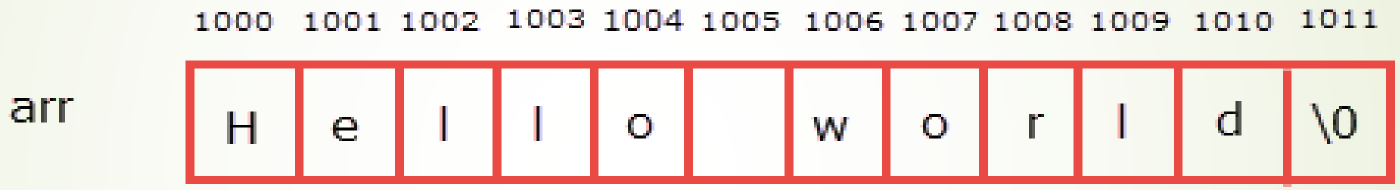
```
int* fun()
{
    int A = 10;
    return (&A);
}

// Driver Code
int main()
{
    // Declare a pointer
    int* p;

    // Function call
    p = fun();

    cout<<p;
    cout<<*p;
    return 0;
}
```

ERRor

```
  int* fun()
{
    // Declare a static integer
    static int A = 10;
    return (&A);
}

// Driver Code
int main()
{
    // Declare a pointer
    int* p;

    // Function call
    p = fun();

    // Print Address
    cout<<p;

    // Print value at the above address
    cout<<*p;
    return 0;
}
```

Character Arrays and pointers

- Character arrays should be large enough to store a string.



12 bytes of memory is allocated to store 12 characters

Functions in string.h

String functions	Description
strcat ( )	Concatenates str2 at the end of str1
strncat ( )	Appends a portion of string to another
strcpy ( )	Copies str2 into str1
strncpy ( )	Copies given number of characters of one string to another
strlen ( )	Gives the length of str1
strcmp ( )	Returns 0 if str1 is same as str2. Returns <0 if strl < str2. Returns >0 if str1 > str2
strcmpi ( )	Same as strcmp() function. But, this function negotiates case. "A" and "a" are treated as same.
strchr ( )	Returns pointer to first occurrence of char in str1
strrchr ( )	last occurrence of given character in a string is found

strstr ( )	Returns pointer to first occurrence of str2 in str1
strrstr ( )	Returns pointer to last occurrence of str2 in str1
strdup ( )	Duplicates the string
strlwr ( )	Converts string to lowercase
strupr ( )	Converts string to uppercase
strrev ( )	Reverses the given string
strset ( )	Sets all character in a string to given character
strnset ( )	It sets the portion of characters in a string to given character
strtok ( )	Tokenizing given string using delimiter

https://www.tutorialspoint.com/c_standard_library/string_h.htm

Character Arrays and pointers

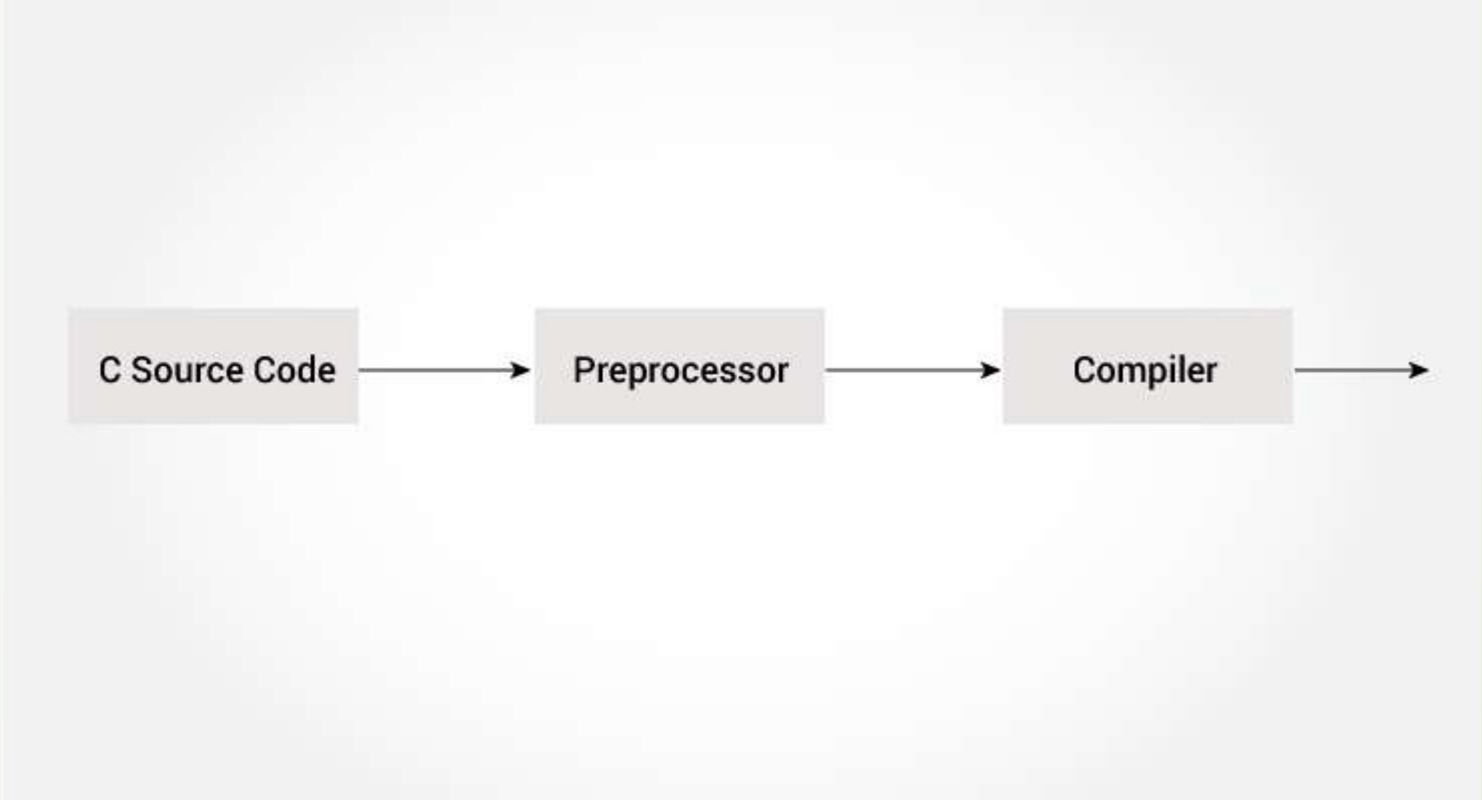
- Like we can handle pointers with array, similarly we can use character type pointers for character arrays.

```
char arr[] = "Hello"; // array version
char *ptr ;// pointer version
ptr=arr ;
```

- We can also print the array as ptr[2];   l
- Ptr[0] = A;   //Aello
- *ptr=A;
- Ptr++…

Macros

- With **macro**(#define): **macro** will be replaced with its **value** in source code compile time only, so compiler does not need to look into memory even single time, **it** compiles code directly with **constant value**. So **it** is better to **use macro** than **constant** variable



Dynamic Memory

- Sometimes the size of the array you declared may be insufficient. To solve this issue, you can allocate memory manually during run-time. This is known as dynamic memory allocation in C programming.

Heap

- Many times, you are not aware in advance how much memory you will need to store particular information in a defined variable and the size of required memory can be determined at run time.
- You can allocate memory at run time within the heap for the variable of a given type using a special operator in C++ which returns the address of the space allocated. This operator is called **new** operator.
- If you are not in need of dynamically allocated memory anymore, you can use **delete** operator, which de-allocates memory that was previously allocated by new operator.

Heap

double* pvalue = NULL; // Pointer initialized with null

pvalue = new double;

```
double* pvalue  = NULL;
if( !(pvalue  = new double )) {
   cout << "Error: out of memory." <<endl;
   exit(1);
}
```

Heap deletion

At any point, when you feel a variable that has been dynamically allocated is not anymore required, you can free up the memory that it occupies in the free store with the ‘delete’ operator as follows −

```
delete pvalue;
```

Dynamic Memory Allocation for Arrays

```
char* pvalue  = NULL;         // Pointer initialized with null
pvalue  = new char[20];
delete [] pvalue;


double** pvalue  = NULL;      // Pointer initialized with null
pvalue  = new double [3][4];
delete [] pvalue;
```

Pointers to 2D array

- In a two dimensional array, we can access each element by using two subscripts, where first subscript represents the row number and second subscript represents the column number.

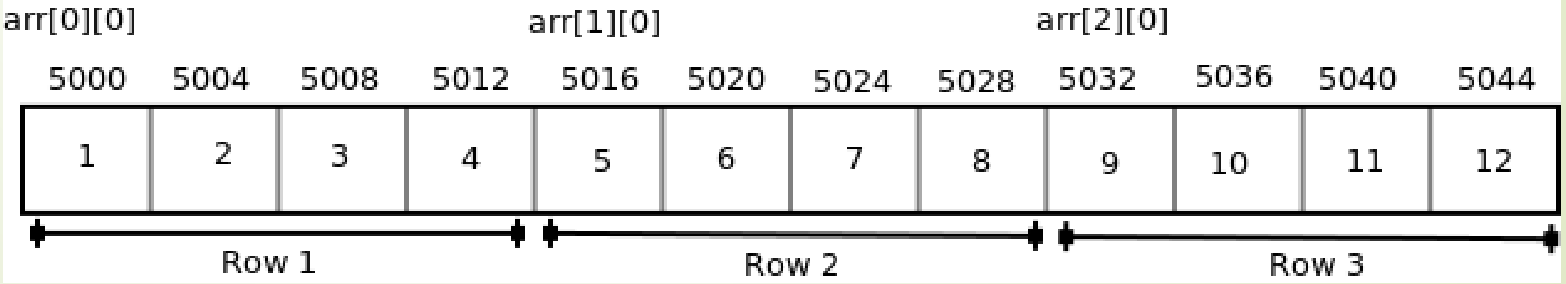
- The elements of 2-D array can be accessed with the help of pointer notation also. Suppose arr is a 2-D array, we can access any element *arr[i][j]* of the array using the pointer expression ***(*(arr + i) + j)**. Now we'll see how this expression can be derived.

Pointers to 2D array

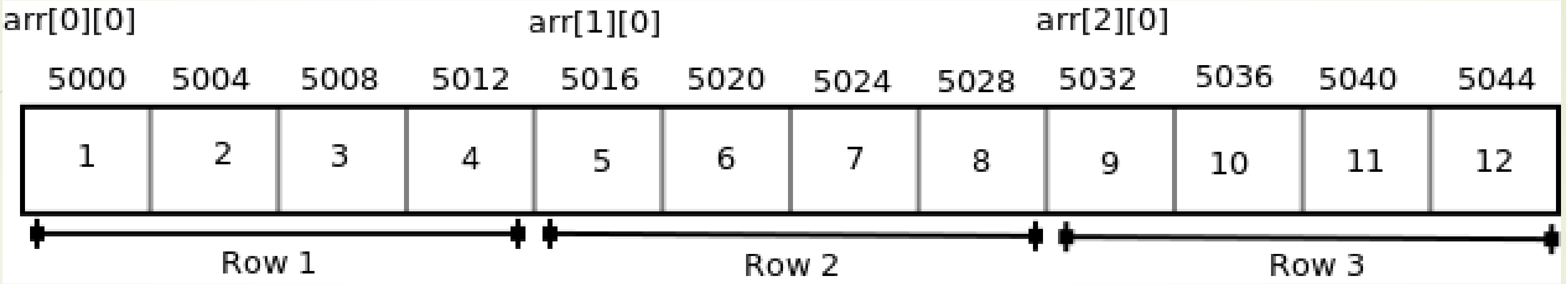
- int arr[3][4] = { {1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12} };

	Col 1	Col 2	Col 3	Col 4
Row 1	1	2	3	4
Row 2	5	6	7	8
Row 3	9	10	11	12

Since memory in a computer is organized linearly it is not possible to store the 2-D array in rows and columns. The concept of rows and columns is only theoretical, actually, a 2-D array is stored in row-major order i.e rows are placed next to each other. The following figure shows how the above 2-D array will be stored in memory.



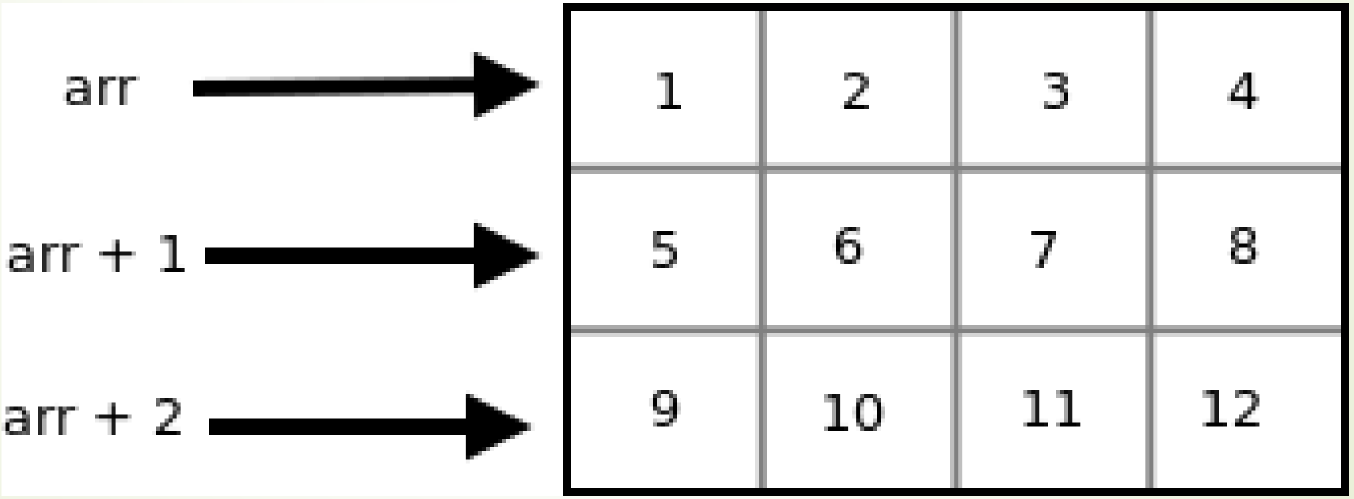
Pointers to 2D array



Each row can be considered as a 1-D array, so a two-dimensional array can be considered as a collection of one-dimensional arrays that are placed one after another.
So here *arr* is an array of 3 elements where each element is a 1-D array of 4 integers.

Pointers to 2D array

Since *arr* is a 'pointer to an array of 4 integers', according to pointer arithmetic the expression arr + 1 will represent the address 5016 and expression arr + 2 will represent address 5032.
So we can say that *arr* points to the 0^{th} 1-D array, *arr + 1* points to the 1^{st} 1-D array and *arr* + 2 points to the 2^{nd} 1-D array.



arr	**-**	**Points to 0^{th} element of arr**	**-**	**Points to 0^{th} 1-D array**	**-**	**5000**
arr + 1	**-**	**Points to 1^{th} element of arr**	**-**	**Points to 1^{nd} 1-D array**	**-**	**5016**
arr + 2	**-**	**Points to 2^{th} element of arr**	**-**	**Points to 2^{nd} 1-D array**	**-**	**5032**

arr	Points to 0^{th} 1-D array
*arr	Points to 0^{th} element of 0^{th} 1-D array
(arr + i)	Points to i^{th} 1-D array
*(arr + i)	Points to 0^{th} element of i^{th} 1-D array
*(arr + i) + j)	Points to j^{th} element of i^{th} 1-D array
((arr + i) + j)	Reprents the value of j^{th} element of i^{th} 1-D array

- To access an individual element of our 2-D array, we should be able to access any jth element of ith 1-D array.

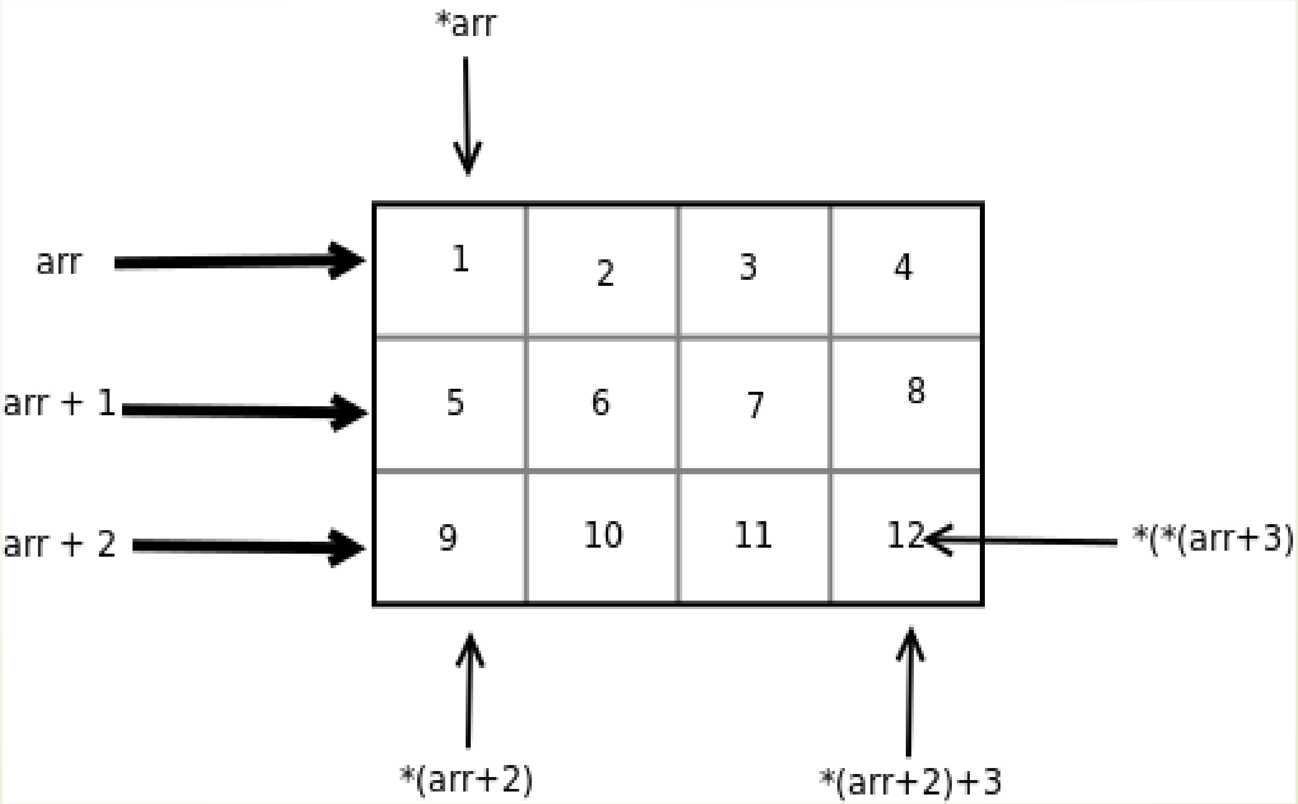
- Since the base type of *(arr + i) is int and it contains the address of 0th element of ith 1-D array, we can get the addresses of subsequent elements in the ith 1-D array by adding integer values to *(arr + i).

- For example *(arr + i) + 1 will represent the address of 1^{st} element of 1^{st} element of ith 1-D array and *(arr+i)+2 will represent the address of 2nd element of ith 1-D array.

- Similarly *(arr + i) + j will represent the address of jth element of ith 1-D array. On dereferencing this expression we can get the jth element of the ith 1-D array.

- For example *(arr + i) + 1 will represent the address of 1st element of 1st element of ith 1-D array and *(arr+i)+2 will represent the address of 2nd element of ith 1-D array.

- Similarly *(arr + i) + j will represent the address of jth element of ith 1-D array. On dereferencing this expression we can get the jth element of the ith 1-D array.



```
int main()
{
  int arr[3][4] = {
                    { 10, 11, 12, 13 },
                    { 20, 21, 22, 23 },
                    { 30, 31, 32, 33 }
                  };
  int i, j;
  for (i = 0; i < 3; i++)
  {
   cout<<  i, arr[i], *(arr + i);

    for (j = 0; j < 4; j++)
      cout<<arr[i][j]<< *(*(arr + i) + j);
    cout<<\n";
  }

  return 0;
```

Address of 0th array = 0x7ffe50edd580
0x7ffe50edd580
10 10 11 11 12 12 13 13
Address of 1th array = 0x7ffe50edd590
0x7ffe50edd590
20 20 21 21 22 22 23 23
Address of 2th array = 0x7ffe50edd5a0
0x7ffe50edd5a0
30 30 31 31 32 32 33 33

```
using namespace std;

// Driver Code
int main()
{
    // Dimensions of the array
    int m = 3, n = 4, c = 0;

    // Declare memory block of size M
    int** a = new int*[m];

    for (int i = 0; i < m; i++) {

        // Declare a memory block
        // of size n
        a[i] = new int[n];
    }

    // Traverse the 2D array
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {

            // Assign values to the
            // memory blocks created
            a[i][j] = ++c;
        }
    }

    // Traverse the 2D array
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {

            // Print the values of
            // memory blocks created
            cout << a[i][j] << " ";
        }
        cout << endl;
    }

    //Delete the array created
    for(int i=0;i<m;i++)    //To delete the inner arrays
    delete [] a[i];
    delete [] a;            //To delete the outer array
                            //which contained the pointers
                            //of all the inner arrays

    return 0;
}
```