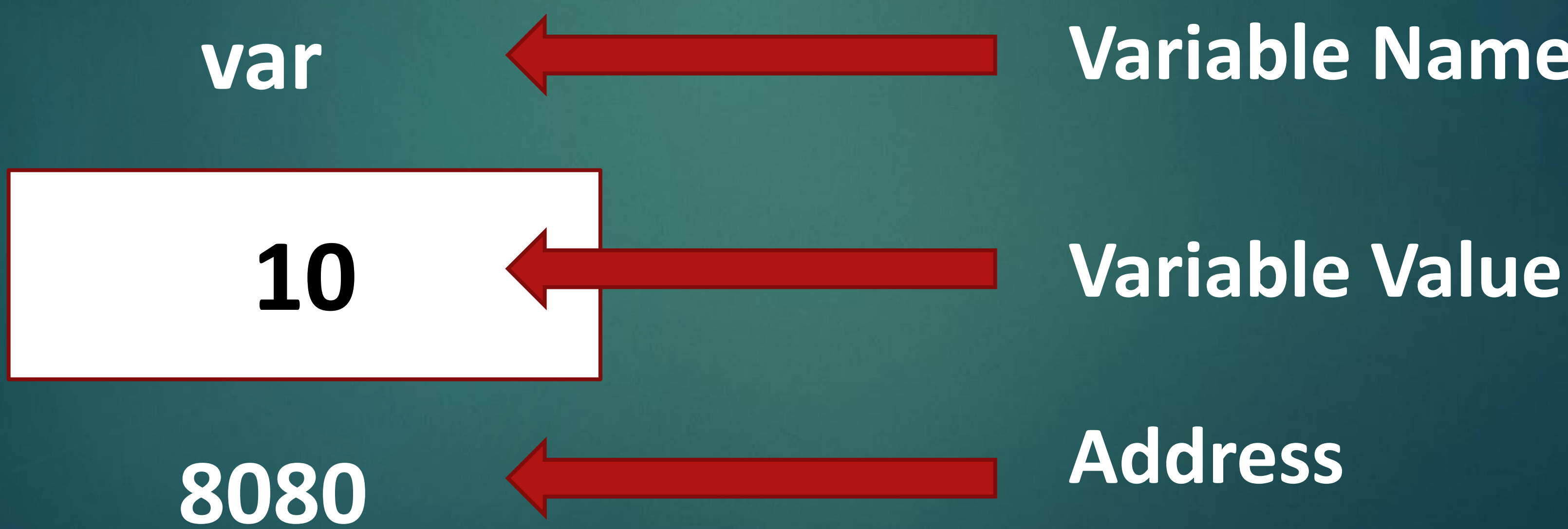
Pointers

Variable

- Variable is named memory location which holds data which can be changed
- Variable has name, address and value.



Variable

- A variable can store address of another variable (As Address is unsigned int)

```
#include<iostream>
using namespace std;
int main()
{
unsigned int addr;
int k =10;
addr =(unsigned int)&k; // Allowed as &k is unsigned int
cout<<addr;
return 0;
}
```

Pointer

- Pointer is variable that stores address of another variable and it can be referenced and dereferenced.
- Three easy steps to use pointer
  - **Declaration of pointer**
  - **Referencing or correct initialization of pointer**
  - **Dereferencing**

Pointer Declaration

- Syntax for pointer declaration

  datatype * ptr_name;

Examples:

  int* p1; // p1 is integer pointer or pointer to int

  float *p2; // p2 is pointer to float

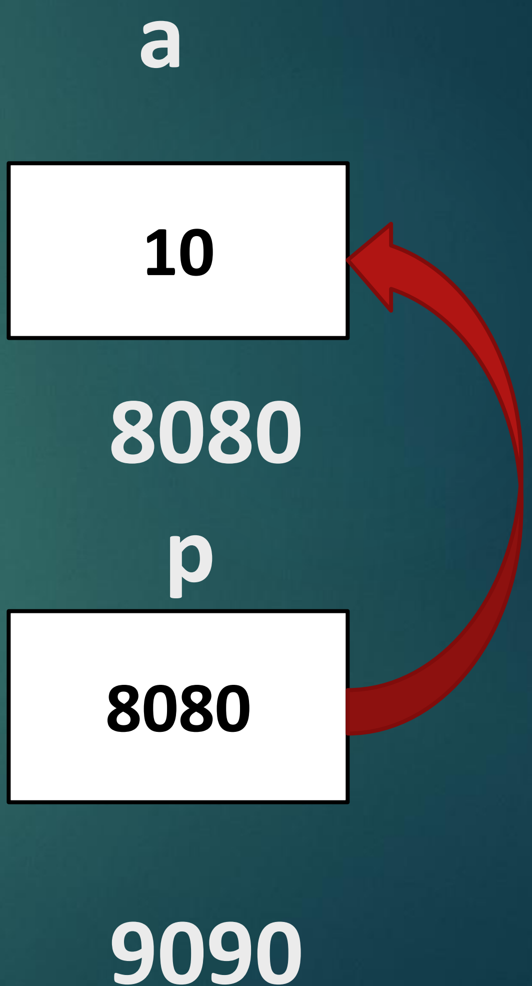
  int *p5, p6; // p5 is pointer to int and p6 is int variable

- Uninitialized pointer contains garbage and become wild pointer
- **Size of any type of pointer is always size of unsigned int**

Pointer Referencing and Dereferencing

- Pointer must be referenced before dereferencing

```
#include<iostream>
using namespace std;
int main(){
        int a =10;
        int *p =&a; // & before variable gives address
        cout<<p<<endl;
        cout<<*p<<endl; // * before ptr gives value
        cout<<&p<<endl;
        return 0;
        }
```



Pointer Initialization

Uninitialized pointer contains garbage and become ***wild*** **pointer.** Dereferencing of such a pointer may give some value or **segmentation fault error**.

```
#include<iostream>
using namespace std;
int main()
{
int *p;
cout<<*p;
return 0;
} // unpredictable output
//Dangerous logical error
```

```
#include<iostream>
using namespace std;
int main()
{
int *p = NULL;
cout<<*p;
return 0;
} // predictable output
// Segmentation fault
```

Need of pointers

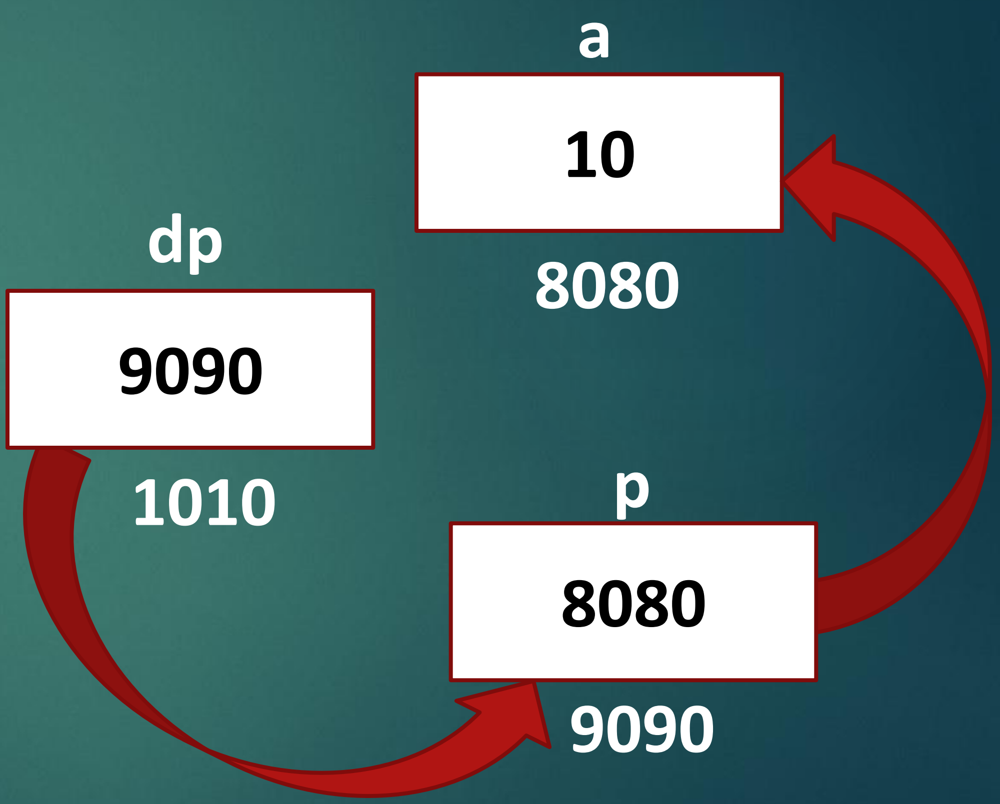
- Access data using address without knowing variable name.
- Return more than one value from function(IN OUT Parameter)
- Access dynamically allocated memory
- Access memory locations within program memory
- Function pointers for callbacks

Double Pointer

Double Pointer (Pointer to pointer)

Example:

```
#include<iostream>
using namespace std;
int main()  {
      int a =10;
      int *p =&a;
//Below is double  pointer
      int **dp=&p;
      cout<<p<<endl;
      cout<<*p<<endl;
      cout<<**dp<<endl;
return 0;}
```



Pointer Conversion

- Any type of pointer can be type casted to any other type with proper type casting.

Example:

```
#include<iostream>
using namespace std;
int main(){
    int k = 65;
     int *ip = &k;
    char *cp =(char*) ip;  // Error without type cast
    cout<<*cp;
return 0; }
```

Generic Pointer(void*)

- Generic pointer can point to any type of variable.
- Generic pointer can not be **dereferenced**
- Pointer arithmetic does not work on generic pointer

```
#include<iostream>
using namespace std;
int main(){
  int k=10;
   int *ip = &k;
  void *vp = ip; // No casting required for generic  pointer
  //cout<<*vp; // Error in dereferencing
  return 0;}
```

Passing by address/reference/pointer

- **Pass by address is used for getting change reflected in actual arguments**

```
#include<iostream>
using namespace std;
void swap(int *pa, int *pb)
{
    int temp = *pa;
    *pa = *pb;
    *pb =temp;
}
```

```
int main()
{
    int a= 10;
    int b= 20;
    swap(&a,&b);
    cout<<"\n a="<<a;
    cout<<"\n b="<<b;
    return 0;
}
```

Returning address/reference/pointer

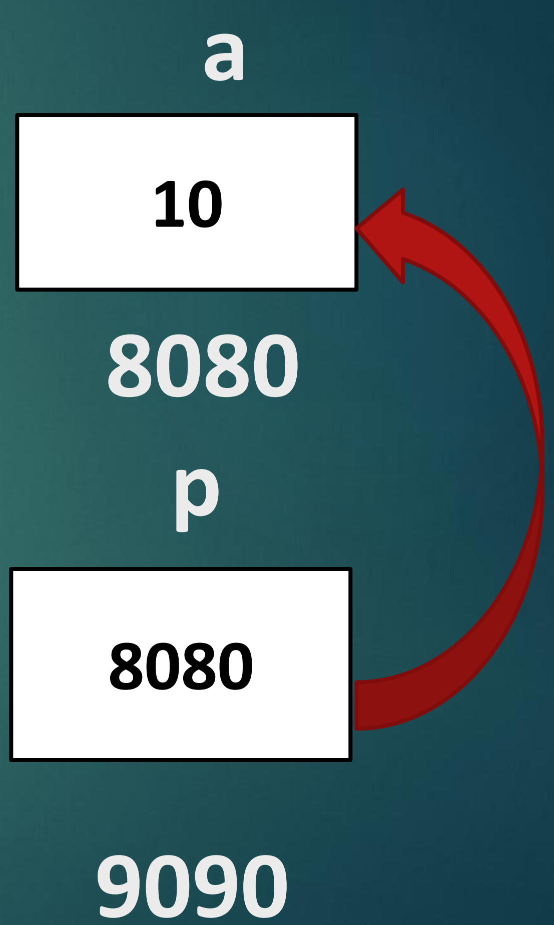
- Returning address of local variable from function may lead to unpredictable output (may get segmentation fault)

```
#include<iostream>
using namespace std;
int k=100;
int* ChangeGlobal()
{
    return &k;
}
int* ChangeLocal()
{ int  i=10;
return &i; // Warning
}
int main(){
 int* p1= ChangeGlobal();
cout<<*p1<<endl;  // print 100
int* p2= ChangeLocal();
//Unpredictable Output
cout<<*p2<<endl;
return 0;
}
```

Pointer Arithmetic

- Pointer can be incremented or decremented.
- Pointer always increment or decrement by **size of data type or one location**

```
#include<iostream>
using namespace std;
int main(){
    int k=10;
    int* p = &k;
    cout<<"\nAddress="<<p;  //8080
    cout<<"\nValue="<<*p;
    p = p+1; // p++;
    cout<<"\nAddress="<<p;  //8084
    return 0;}
```



Pointer Arithmetic

- Integer constant can be added in pointer

  Address = Address + int const

  Address = Address - int const

- Two pointers can not be added but subtracted

```
int arr[3]={1,2,3};
int *p1 =&arr[0];
int *p2 = &arr[2];
int  locations = p2-p1;
cout<<locations;
```

- Pointer arithmetic does not work with void* (generic pointer)
- **void pointer can be created but void type variable can not be created**

Pointers and Constants

- **Pointer to constant**

  Value pointed by pointer to constant can not be changed but pointer can changed

```
#include<iostream>
using namespace std;
int main(){
const int k=20;
const int *p = &k;
cout<<*p;
p = NULL; // can be changed
p = p+1; // can be changed
//*p =100; // Error
return 0;}
```

Pointers and Constants

- **Constant Pointer to variable**

  Value pointed by constant pointer can be changed but pointer can not changed

```
#include<iostream>
using namespace std;
int main(){
	int k=20;
	int * const p = &k;
	cout<<*p<<endl;
	//p = NULL;// can  not be
	   changed
	//p = p+1; // can  not be changed
	*p =100; // Can be changed
	   cout<<*p<<endl;
	return 0;
} // Red colour indicate error
```

Pointers and Constants

- **Constant Pointer**
  - Array name is internally a constant pointer to first element of array and it stores base address of array
  - Function name is internally constant pointer

```
#include<iostream>
using namespace std;
int add(int a, int b)
{  return a+b; }

int main(){
char name[50]= "Priyanka";
name = "Deepika";
name++;
add = add+1;
cout<<name;
return 0;

} // Red colour indicate Error
```

Pointers and Constants

- **Constant Pointer to constant**

  Value pointed by constant pointer to constant can not be changed also pointer can not be changed

```
#include<iostream>
using namespace std;
int main(){
const int k=20;
const int *const p = &k;
cout<<*p<<endl;
p = NULL;// Can  not be changed
p = p+1; // Can  not be changed
*p =100; // Can be changed
cout<<*p<<endl;
return 0;
} // Red colour indicate error
```

Character Pointer

- **Character pointer points to one character and also it can be used for string handling**

```
#include<iostream>
using namespace std;
int main(){
     char c='A';
     char *cp = &c;
     cout<<*cp; //Prints A
return 0;}
```

```
#include<iostream>
using namespace std;
int main(){
  char *name = "Priyanka";
  cout<<name<<endl;
  name = "Deepika";
  cout<<name<<endl;
return 0;}
```

Passing Array to function using Pointer

- Array can be passed to function using array.
- Size or number of elements need to be passed explicitly.

```
#include<iostream>
using namespace std;
void PrintArray(int* p ,int size){
for(int i =0; i< size; i++)
    { cout<<"\n"<<p[i]; //*(p+i)
    }
}
int main(){
 int arr[5] ={1,2,3};
 PrintArray(arr,3);
 return 0;}
```

Passing char array to function using Pointer

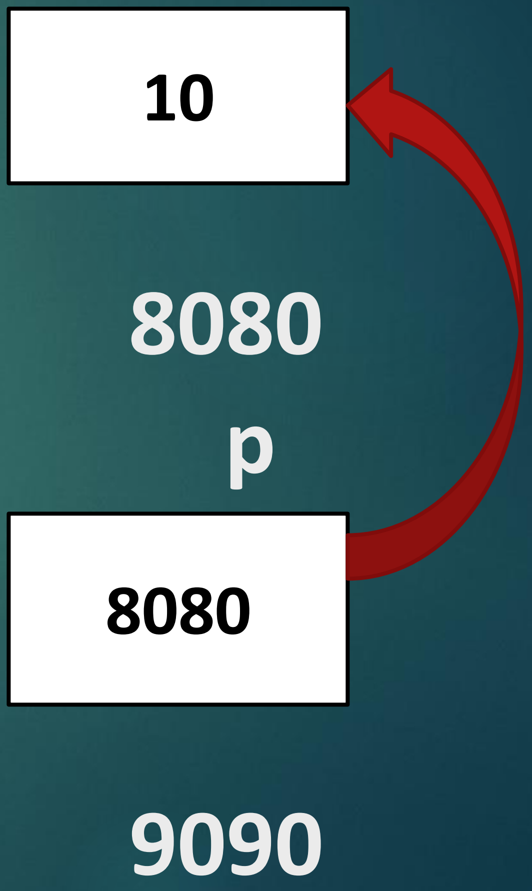
- Character array has **termination character as ‘\0’** hence no need to pass size of array while passing to it to function.

```
#include<iostream>
using namespace std;
void PrintChars(char* p)
{ for(int i =0; p[i] !='\0'; i++)
      { cout<<"\n"<<p[i]; //*(p+i)
      }
}
int main(){
char arr[10] = "Sidhhi";
 PrintChars(arr);
 return 0;}
```

Dynamic Memory Allocation and De-allocation

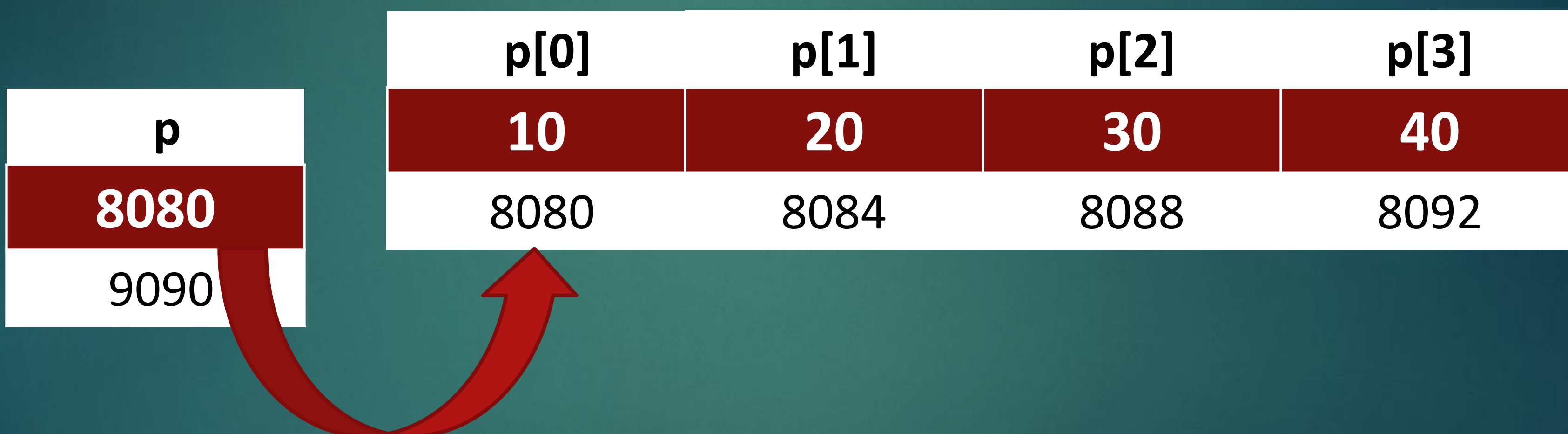
- In C++, dynamic memory allocation is done using new operation and memory is freed using delete operator

```
#include<iostream>
using namespace std;
int main(){
    int *p = new int;
    *p = 10;
    cout<<*p;
    delete p;
return 0;} // Dynamically allocated memory has no variable name
```



Dynamic Memory Allocation for Array

- For an array memory can be allocated using new and de-allocated using delete operator



Dynamic Memory Allocation for Array

Example:

```
#include<iostream>
using namespace std;
int main(){
    int *p = new int[4]; // Allocation of memory for array
    for(int i =0; i<4; i++){//  Loop for initializing array elements
    p[i]=i*10; }
    for(int i =0; i<4; i++){ //  Loop for printing array elements
    cout<< p[i]<<endl; }
    delete []p;  // De-allocation of memory
return 0;}
```

Array of Pointers

- Array is called as derived data type as it is collection of elements of same types
- Pointer is also a derived data type hence Array of Pointers can be created.
- Syntax:

  <datatype>* <arr_name>[size];

  <> Denotes placeholders

Ex. **int *ptrArr[10];** // Each element will store address

Array of Pointers


```
#include<iostream>
using namespace std;
int main(){
    char* names[3];
    names[0] = "Priyanka";
    names[1] = "Deepika";
    names[2] = "Kareena";
    for(int i=0; i<3;i++)
    { cout<<"\n"<<
names[i];}
return 0;}
```

// Initially each char* in array will point to garbage

//We need to initialize all the pointers in array

//names[0] is character pointer

Pointer to array

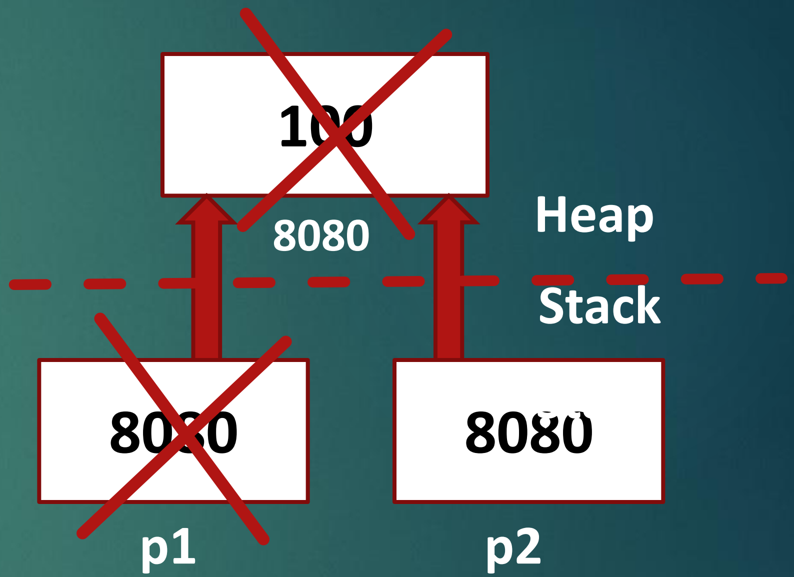
- Pointer to array will point to entire array and increment or decrement by one whole array

```
#include<iostream>
using namespace std;
int main(){
    char arr[3][10] ={"Bebo","Piggy","Deeps"};
    char (*parr)[10] =&arr[0];  // parr is ptr to char array of size 10
    cout<<*parr<<endl; // will print Bebo
    parr = parr +2;  // Will increment by two array
    cout<<*parr<<endl;  // Will print Deeps
return 0;}
```

Dangling Pointer

- Dangling pointer is pointer which pointing to a memory location which is invalid and already freed by somebody

```
#include<iostream>
using namespace std;
int main(){
    int *p1 = new int;
    int *p2 = p1;
    *p2=100;
    cout<<*p1<<endl;
    delete p1;
    cout<<*p2<<endl; // Segmentation fault // p2 is dangling pointer
return 0;} //  p2 is trying to access  memory deleted by p1
```



Memory Leak

- If programmer allocates memory dynamically, it is programmers responsibility to delete memory. The memory allocated dynamically but not deleted/ freed is called as **MEMORY LEAK**

```
#include<iostream>
using namespace std;
int main(){
    int *p = new int;
    *p = 10;
    cout<<*p;
return 0;} // Here programmer forgot to free memory and it becomes memory leak
```

Lvalue and Rvalue

- **lvalue** (locator value) represents an object that occupies some identifiable location in memory (i.e. has an address).
- **rvalues** is some value that can be assigned and used on RHS of == operator

Example:

int i =4; **// 4 is rvalue & i is lvalue**

char arr[100]="Priyanaka";

4 = i; **// Lvalue required error**

arr ="Deepika"; **//Lvalue required error**

Thank You

Remember me !!!!! ……….POINTER