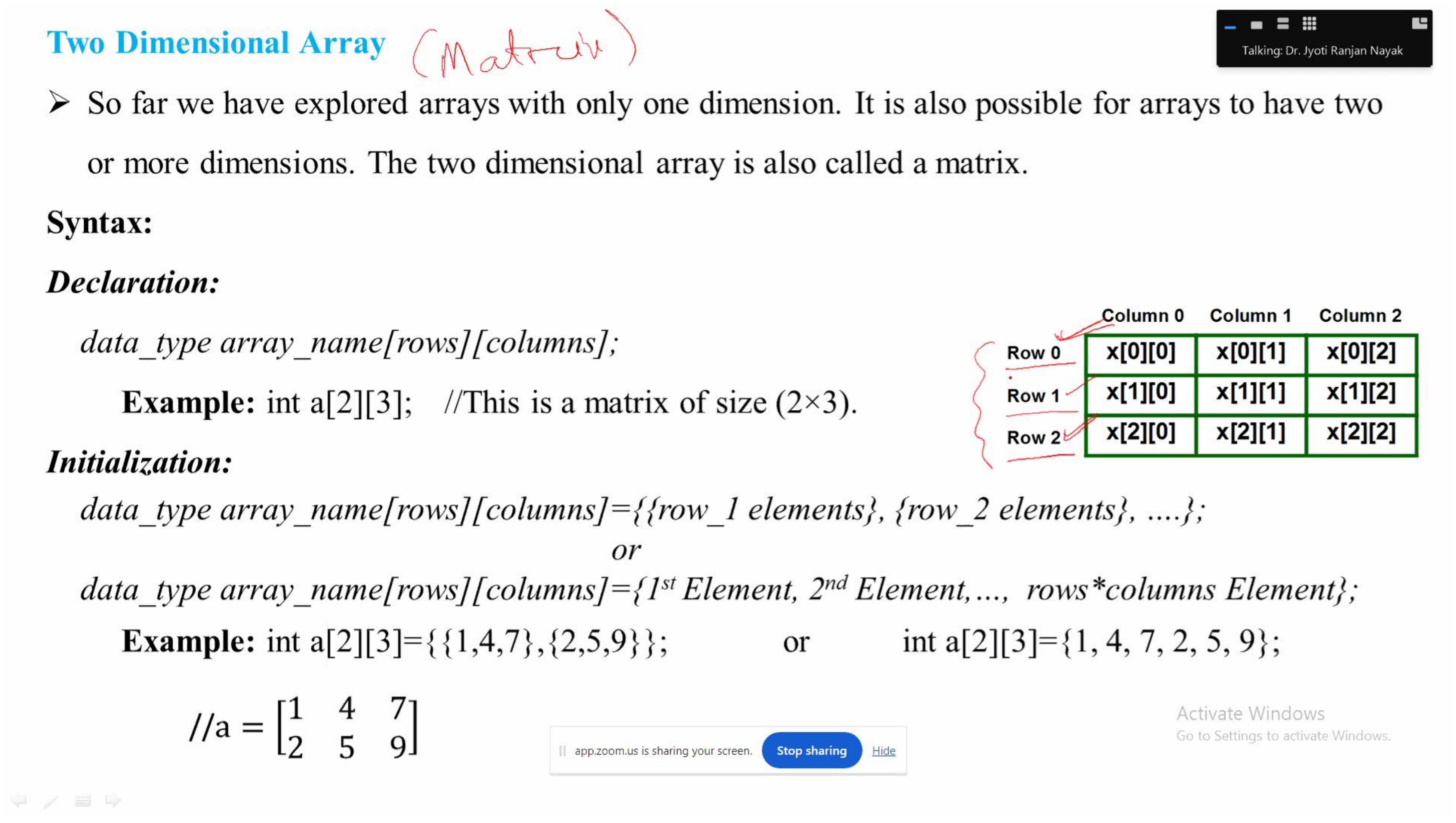


Two Dimensional Array (Matrix)

- So far we have explored arrays with only one dimension. It is also possible for arrays to have two or more dimensions. The two dimensional array is also called a matrix.

Syntax:

Declaration:

data_type array_name[rows][columns];

Example: int a[2][3]; //This is a matrix of size (2×3).

	Column 0	Column 1	Column 2
Row 0	x[0][0]	x[0][1]	x[0][2]
Row 1	x[1][0]	x[1][1]	x[1][2]
Row 2	x[2][0]	x[2][1]	x[2][2]

Initialization:

data_type array_name[rows][columns]={{row_1 elements}, {row_2 elements}, ...};

or

*data_type array_name[rows][columns]={1st Element, 2nd Element,..., rows*columns Element};*

Example: int a[2][3]={{1,4,7},{2,5,9}}; or int a[2][3]={1, 4, 7, 2, 5, 9};

$$//a = \begin{bmatrix}1 & 4 & 7\\ 2 & 5 & 9\end{bmatrix}$$

```
#include<iostream>
using namespace std;
int main()
{
   int a[2][3]={{1,4,6},{2,6,8}};
   int i,j;
   for(i=0;i<2;i++)
   {
      for(j=0;j<3;j++)
      {
         cout<<a[i][j]<<'\t';
      }
      cout<<endl;
   }
   return(0);
}
```

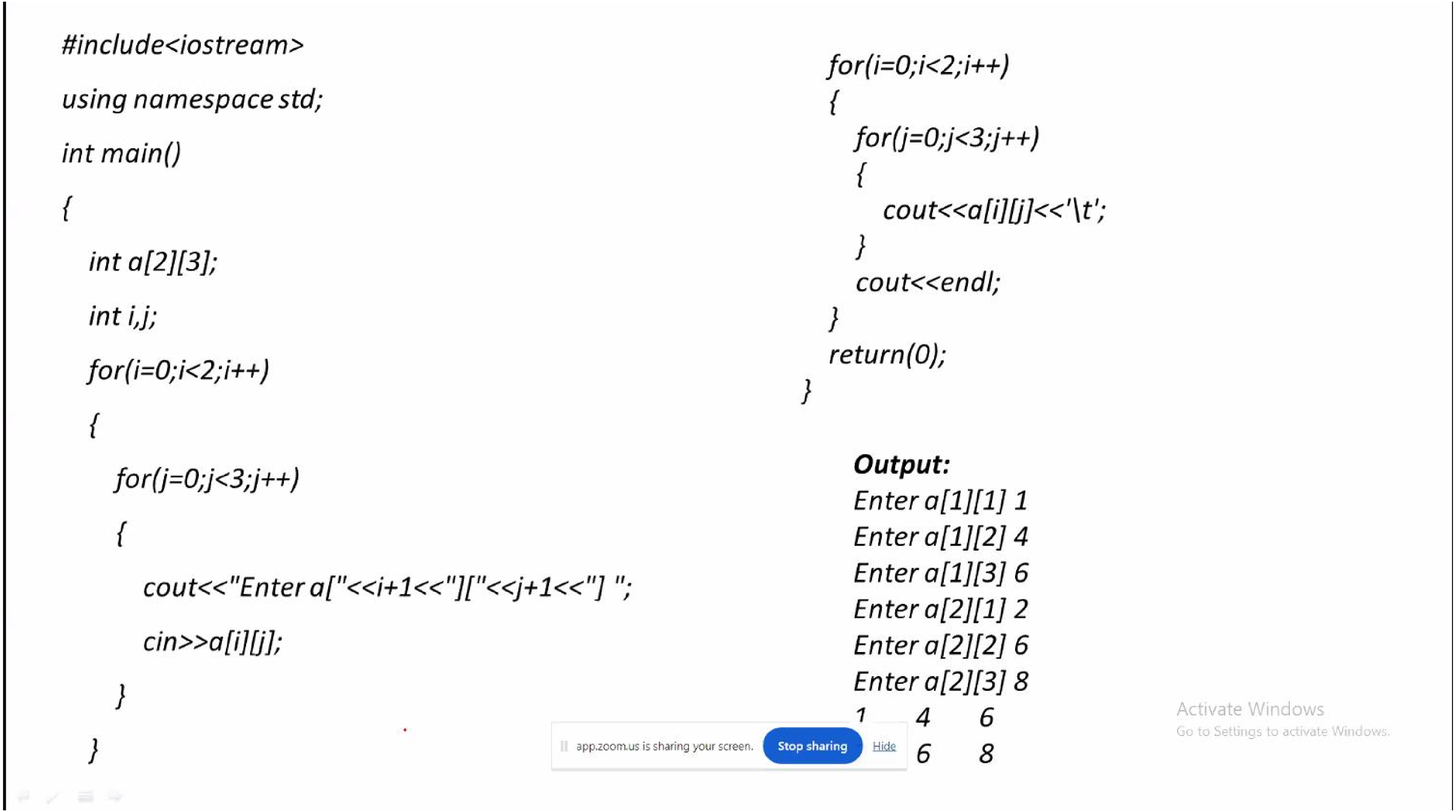
Output:

```
1    4    6
2    6    8
```

```
#include<iostream>
using namespace std;
int main()
{
   int a[2][3]={1,4,6,2,6,8};
   int i,j;
   for(i=0;i<2;i++)
   {
      for(j=0;j<3;j++)
      {
         cout<<a[i][j]<<'\t';
      }
      cout<<endl;
   }
   return(0);
}
```

Output:

```
1    4    6
2    6    8
```



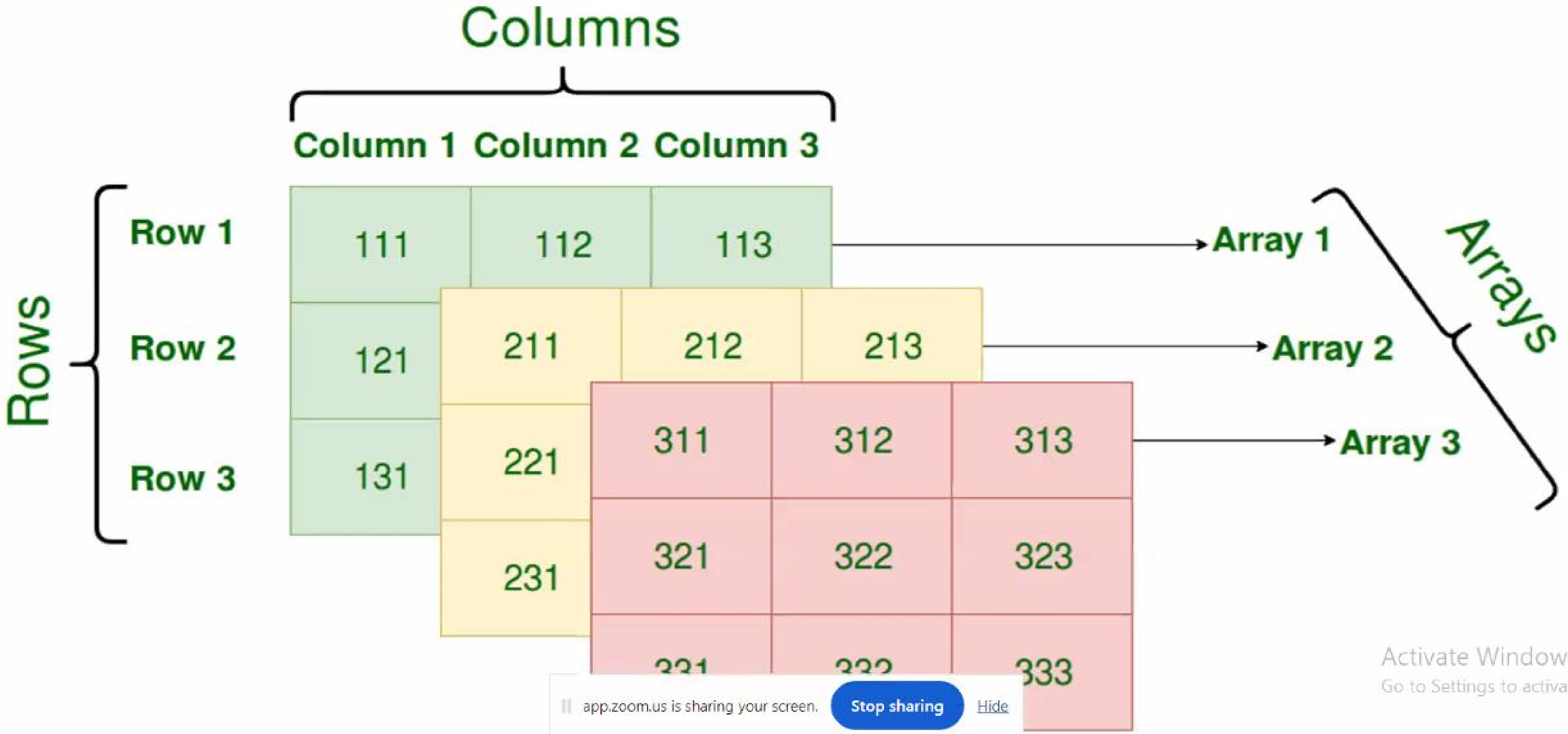
```
#include<iostream>
using namespace std;
int main()
{
   int a[2][3];
   int i,j;
   for(i=0;i<2;i++)
   {
      for(j=0;j<3;j++)
      {
         cout<<"Enter a["<<i+1<<"]["<<j+1<<"] ";
         cin>>a[i][j];
      }
   }
   for(i=0;i<2;i++)
   {
      for(j=0;j<3;j++)
      {
         cout<<a[i][j]<<'\t';
      }
      cout<<endl;
   }
   return(0);
}
```

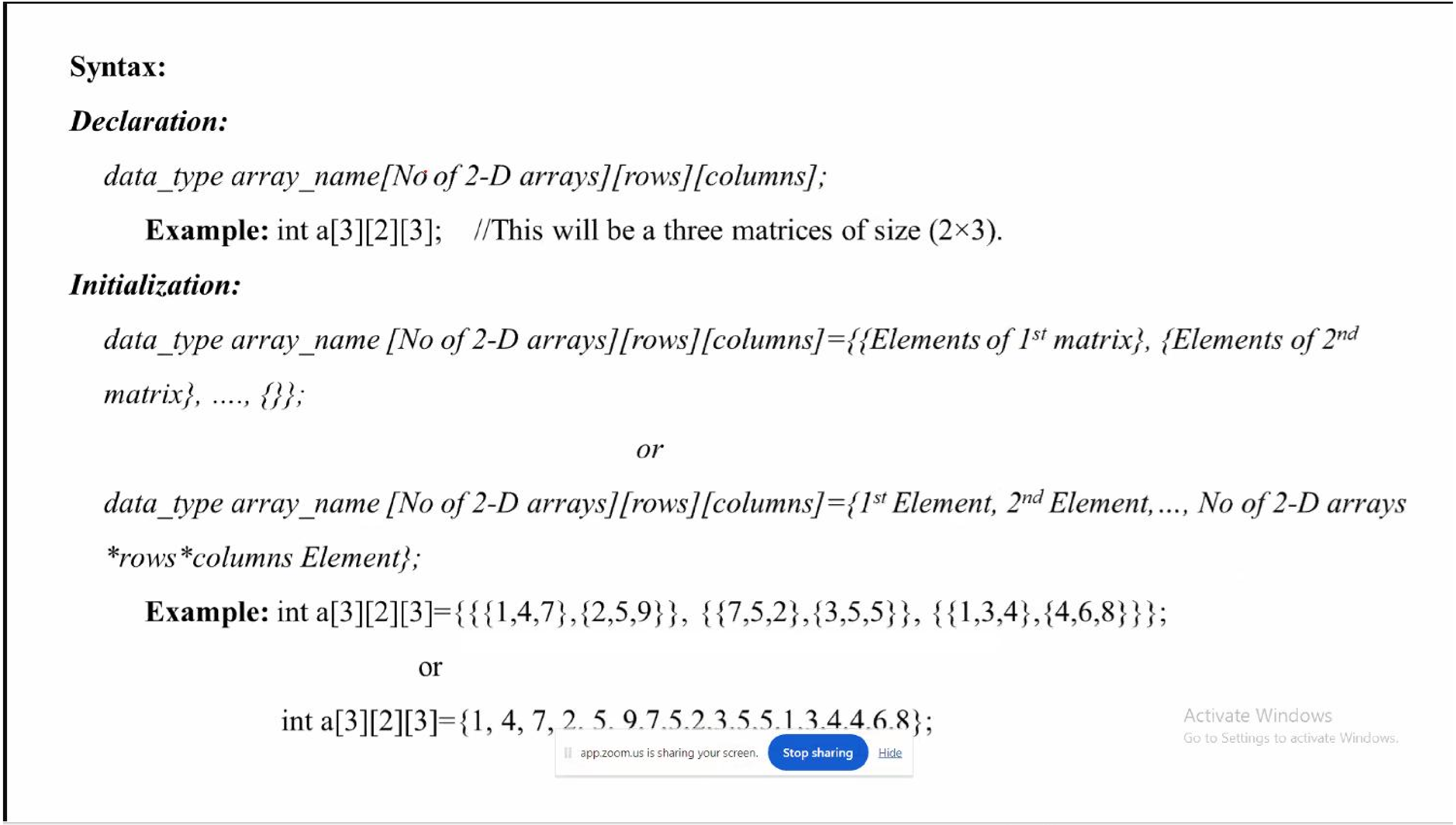
Output:

```
Enter a[1][1] 1
Enter a[1][2] 4
Enter a[1][3] 6
Enter a[2][1] 2
Enter a[2][2] 6
Enter a[2][3] 8
1       4       6
        6       8
```

Multi Dimensional Array

- A Three Dimensional Array or 3D array in C++ is a collection of two-dimensional arrays. It can be visualized as multiple 2D arrays stacked on top of each other.





Syntax:

Declaration:

data_type array_name[No of 2-D arrays][rows][columns];

Example: int a[3][2][3]; //This will be a three matrices of size (2×3).

Initialization:

data_type array_name [No of 2-D arrays][rows][columns]={{Elements of 1st matrix}, {Elements of 2nd matrix}, ..., {}};

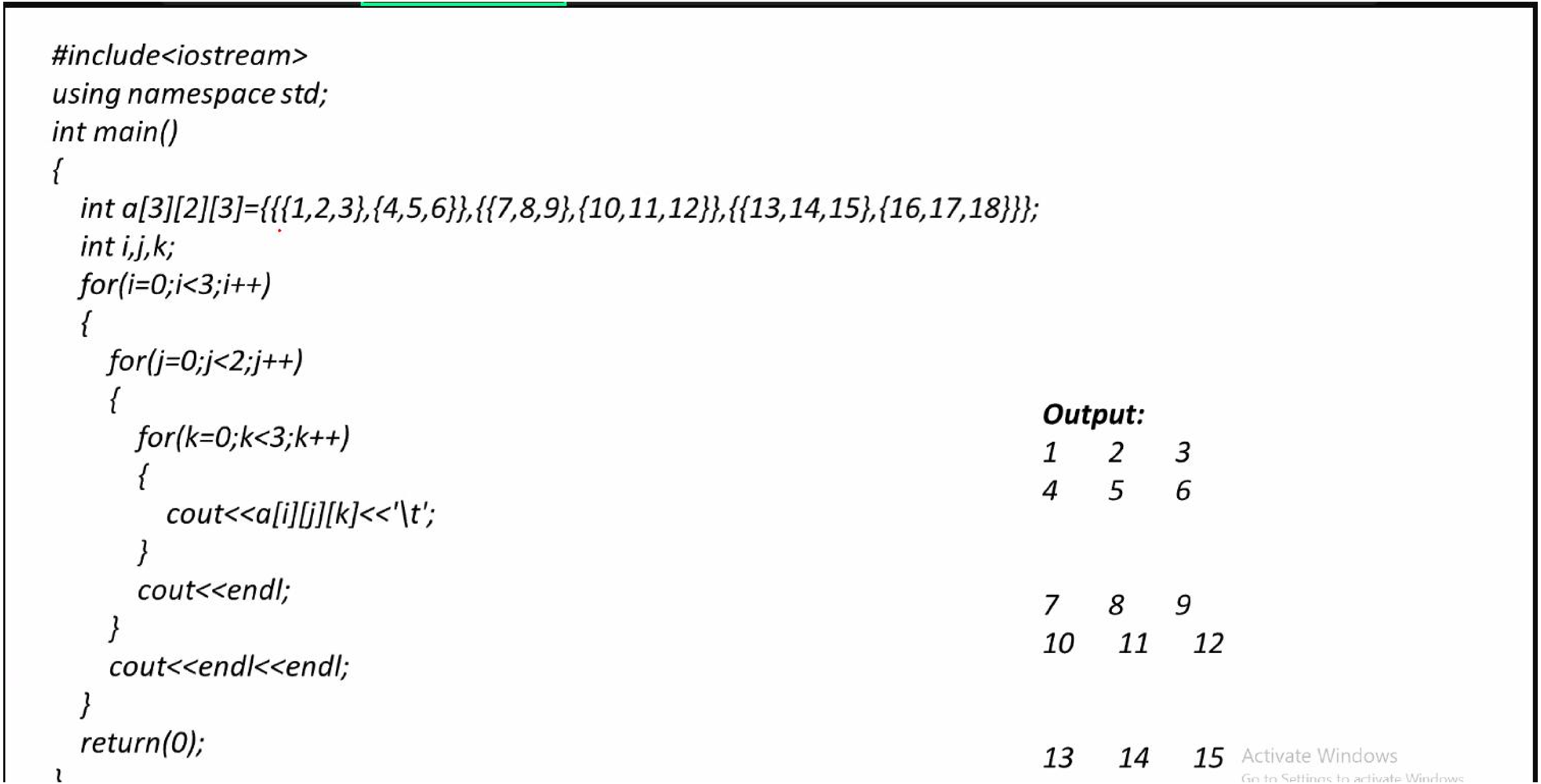
or

*data_type array_name [No of 2-D arrays][rows][columns]={1st Element, 2nd Element, ..., No of 2-D arrays *rows*columns Element};*

Example: int a[3][2][3]={{{1,4,7},{2,5,9}}, {{7,5,2},{3,5,5}}, {{1,3,4},{4,6,8}}};

or

int a[3][2][3]={1, 4, 7, 2, 5, 9,7,5,2,3,5,5,1,3,4,4,6,8};



```
#include<iostream>
using namespace std;
int main()
{
   int a[3][2][3]={{{1,2,3},{4,5,6}},{{7,8,9},{10,11,12}},{{13,14,15},{16,17,18}}};
   int i,j,k;
   for(i=0;i<3;i++)
   {
      for(j=0;j<2;j++)
      {
         for(k=0;k<3;k++)
         {
            cout<<a[i][j][k]<<'\t';
         }
         cout<<endl;
      }
      cout<<endl<<endl;
   }
   return(0);
}
```

Output:

```
1      2      3
4      5      6


7      8      9
10     11     12


13     14     15
```

Pointer

- A pointer is a variable that holds the address of a variable or a function. A pointer is a powerful feature that adds enormous power and flexibility to C++ language.
- If you have a variable *var* in your program, *&var* will give you its address in the memory.
- As the pointers in C++ store the memory addresses, their size is independent of the type of data they are pointing to.

Pointer Declaration:

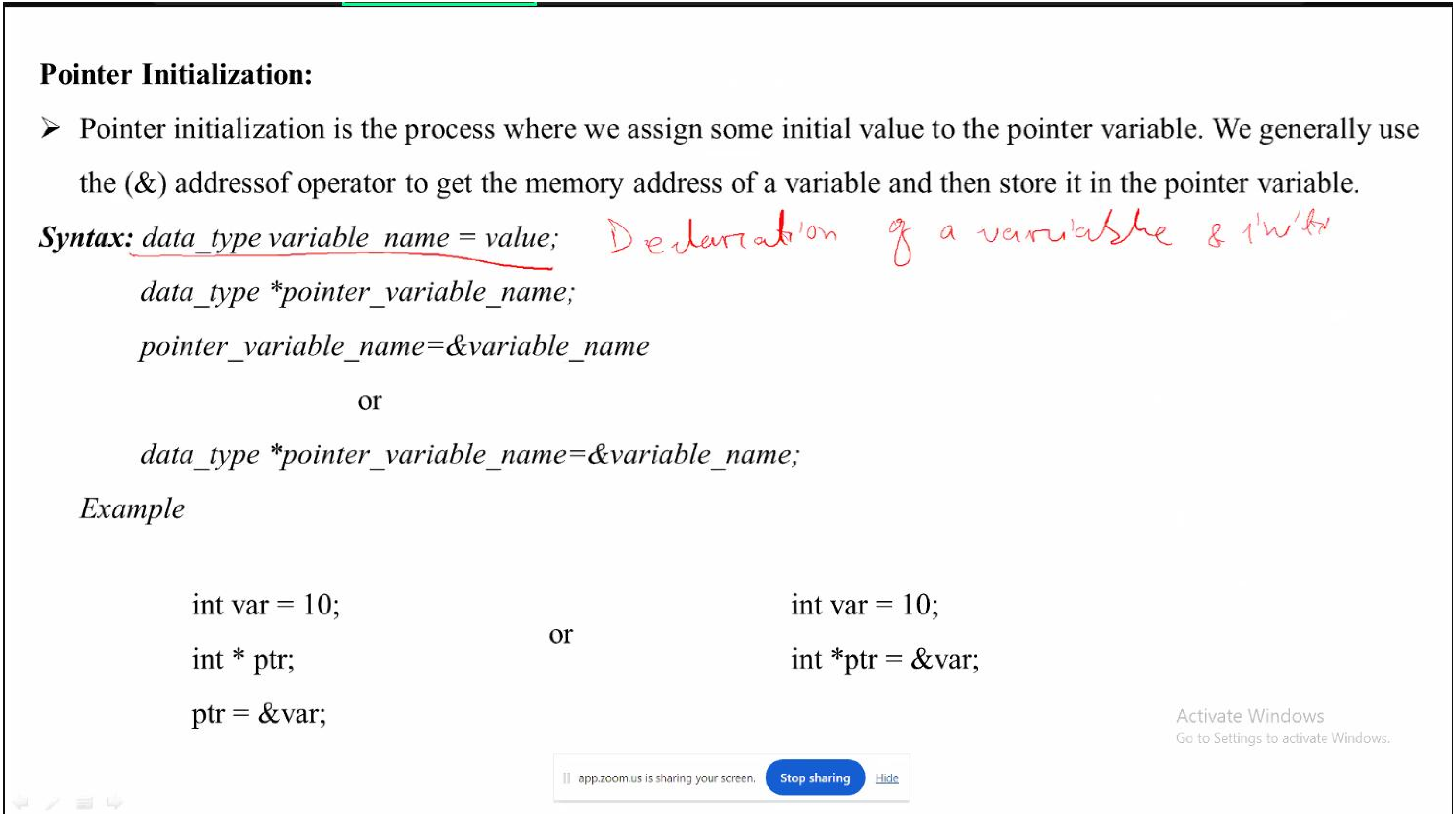
- To declare a pointer, we use the (*) dereference operator before its name.

Syntax: *data_type *pointer_variable_name;*

Example

*int *ptr; //ptr is the pointer variable and *ptr holds the values stored in the variable.*

- The pointer declared here will point to some random memory address as it is not initialized. Such pointers are called *wild pointers*.



Pointer Initialization:

- Pointer initialization is the process where we assign some initial value to the pointer variable. We generally use the (&) addressof operator to get the memory address of a variable and then store it in the pointer variable.

Syntax: *data_type variable_name = value;* Declaration of a variable & initz

*data_type *pointer_variable_name;*

pointer_variable_name=&variable_name

or

*data_type *pointer_variable_name=&variable_name;*

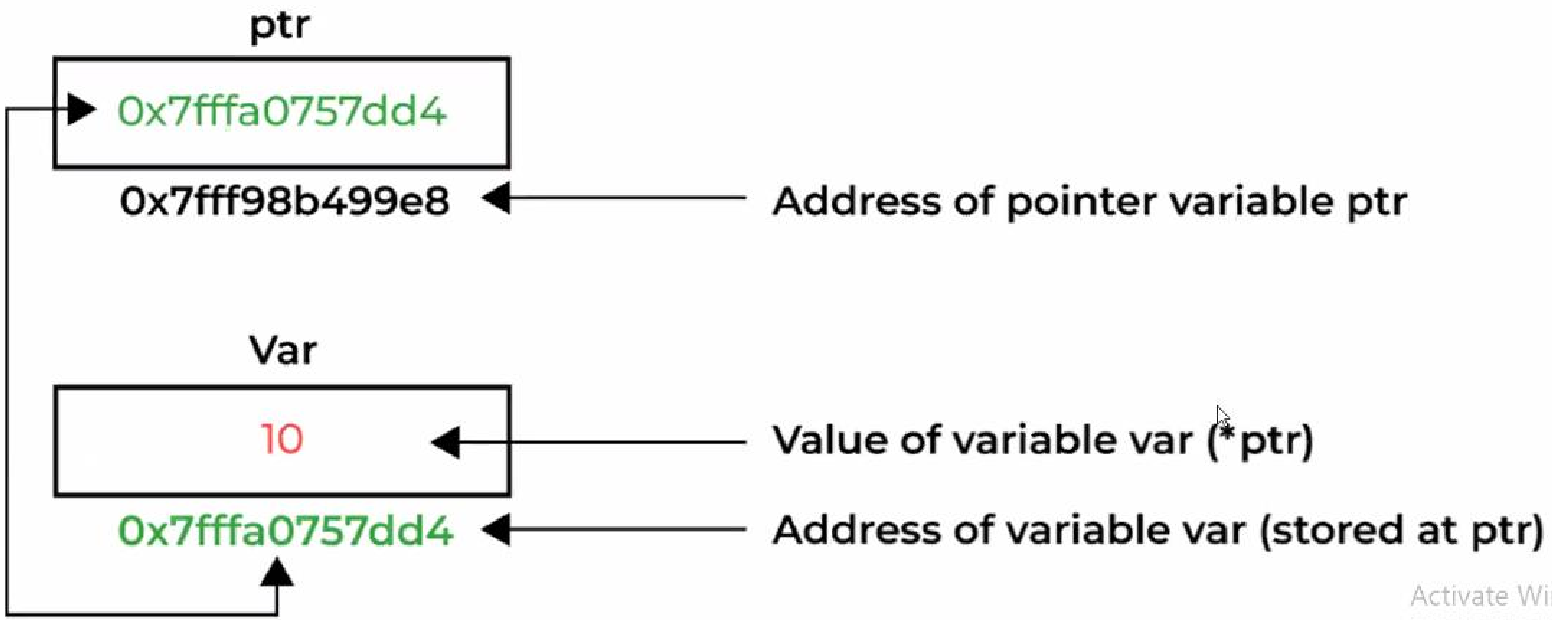
Example

```
int var = 10;                  int var = 10;
int * ptr;          or         int *ptr = &var;
ptr = &var;
```

Activate Windows
Go to Settings to activate Windows.
app.zoom.us is sharing your screen.
Stop sharing
Hide

Pointer Dereferencing

- Dereferencing a pointer is the process of accessing the value stored in the memory address specified in the pointer.
- We use the same **(*) dereferencing operator** that we used in the pointer declaration.



ptr

0x7fffa0757dd4

0x7fff98b499e8

Address of pointer variable ptr

Var

10

Value of variable var (*ptr)

0x7fffa0757dd4

Address of variable var (stored at ptr)

Example:

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

Output:

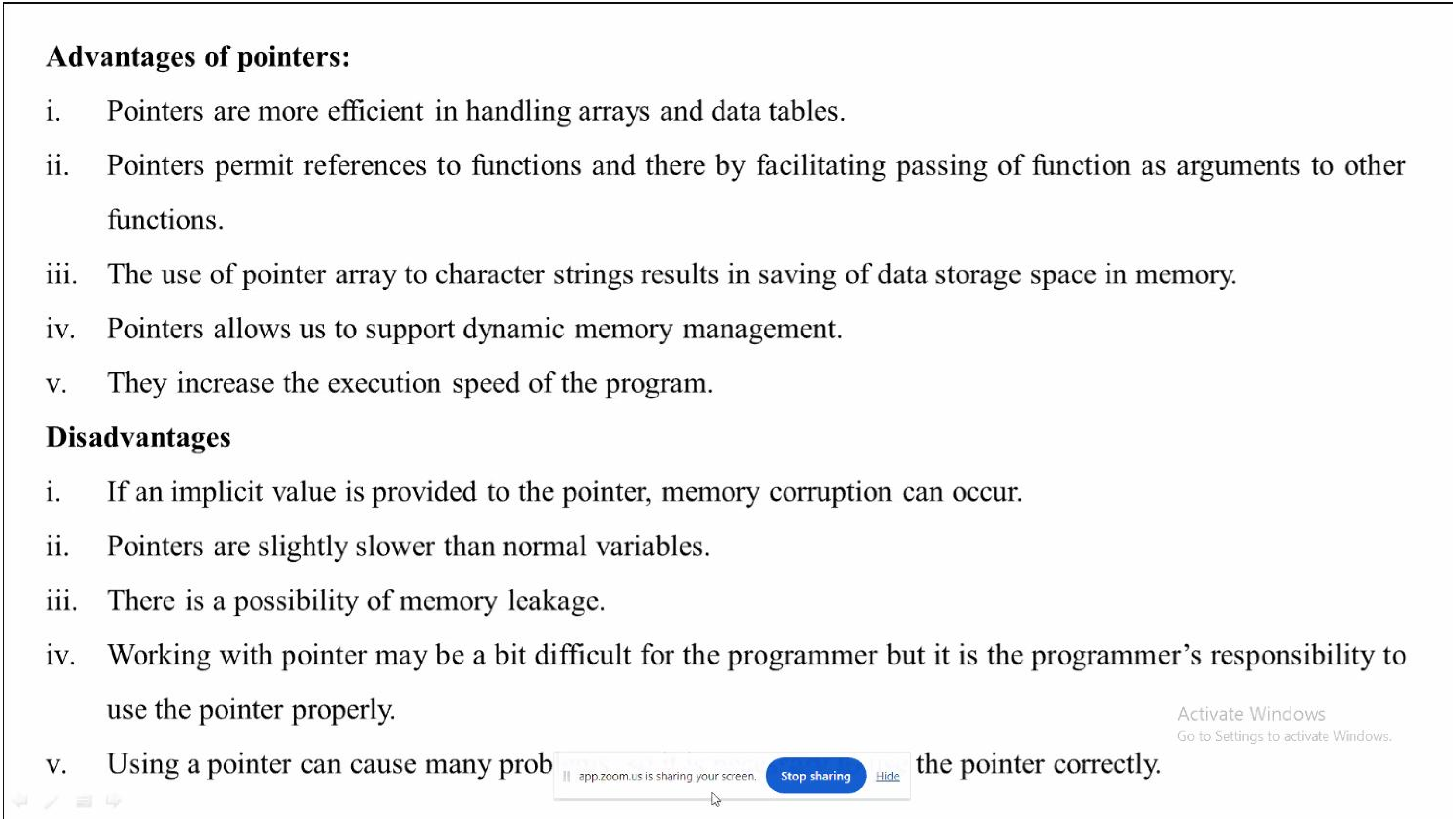
```
10
0x61ff08
0x61ff08
10
```

Example

```
#include<iostream>
using namespace std;
int main()
{
   int a=10;
   int *p=&a;
   cout<<a<<endl;
   cout<<p<<endl;
   cout<<*p<<endl;
   *p=22;
   cout<<a<<endl;
   cout<<p<<endl;
   cout<<*p<<endl;
   return(0);
}
```

Output:

```
10
0x61ff08
10
22
0x61ff08
22
```



Advantages of pointers:

i. Pointers are more efficient in handling arrays and data tables.

ii. Pointers permit references to functions and there by facilitating passing of function as arguments to other functions.

iii. The use of pointer array to character strings results in saving of data storage space in memory.

iv. Pointers allows us to support dynamic memory management.

v. They increase the execution speed of the program.

Disadvantages

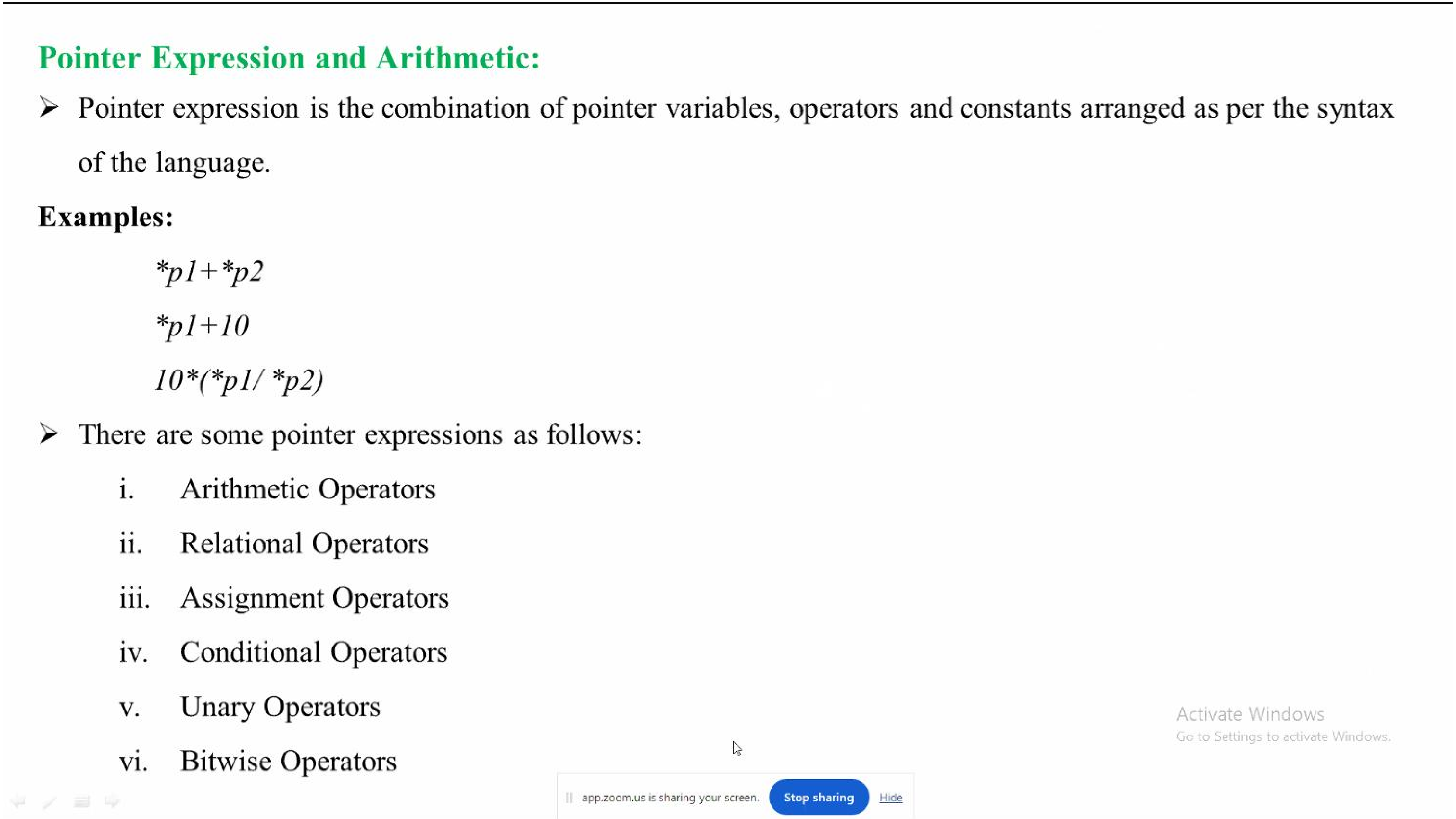
i. If an implicit value is provided to the pointer, memory corruption can occur.

ii. Pointers are slightly slower than normal variables.

iii. There is a possibility of memory leakage.

iv. Working with pointer may be a bit difficult for the programmer but it is the programmer's responsibility to use the pointer properly.

v. Using a pointer can cause many prob the pointer correctly.



Pointer Expression and Arithmetic:

- Pointer expression is the combination of pointer variables, operators and constants arranged as per the syntax of the language.

Examples:

$*p1+*p2$

$*p1+10$

$10*(*p1/ *p2)$

- There are some pointer expressions as follows:

    i. Arithmetic Operators

    ii. Relational Operators

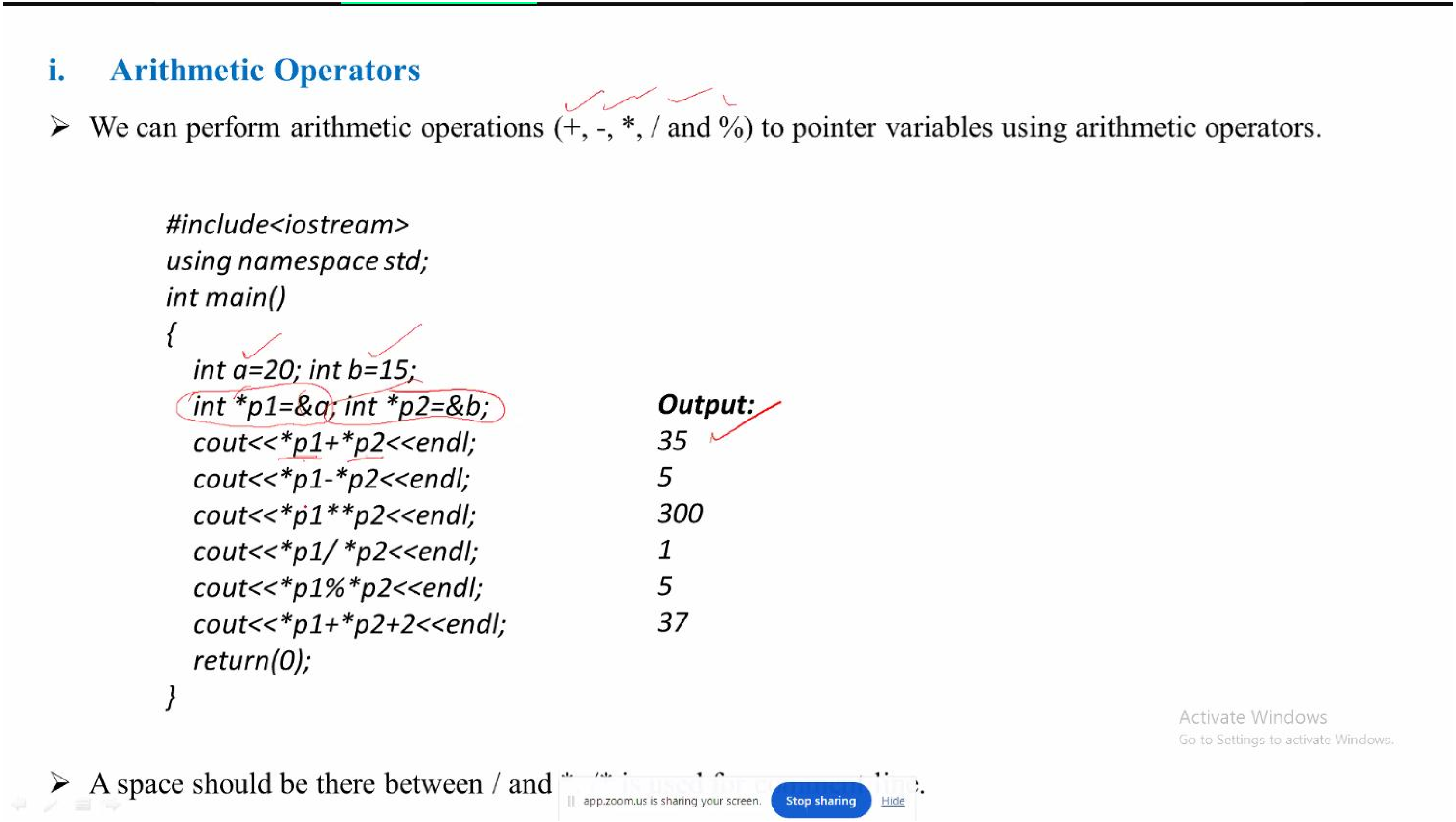
    iii. Assignment Operators

    iv. Conditional Operators

    v. Unary Operators

    vi. Bitwise Operators

Activate Windows
Go to Settings to activate Windows.
app.zoom.us is sharing your screen.
Stop sharing
Hide



i. Arithmetic Operators

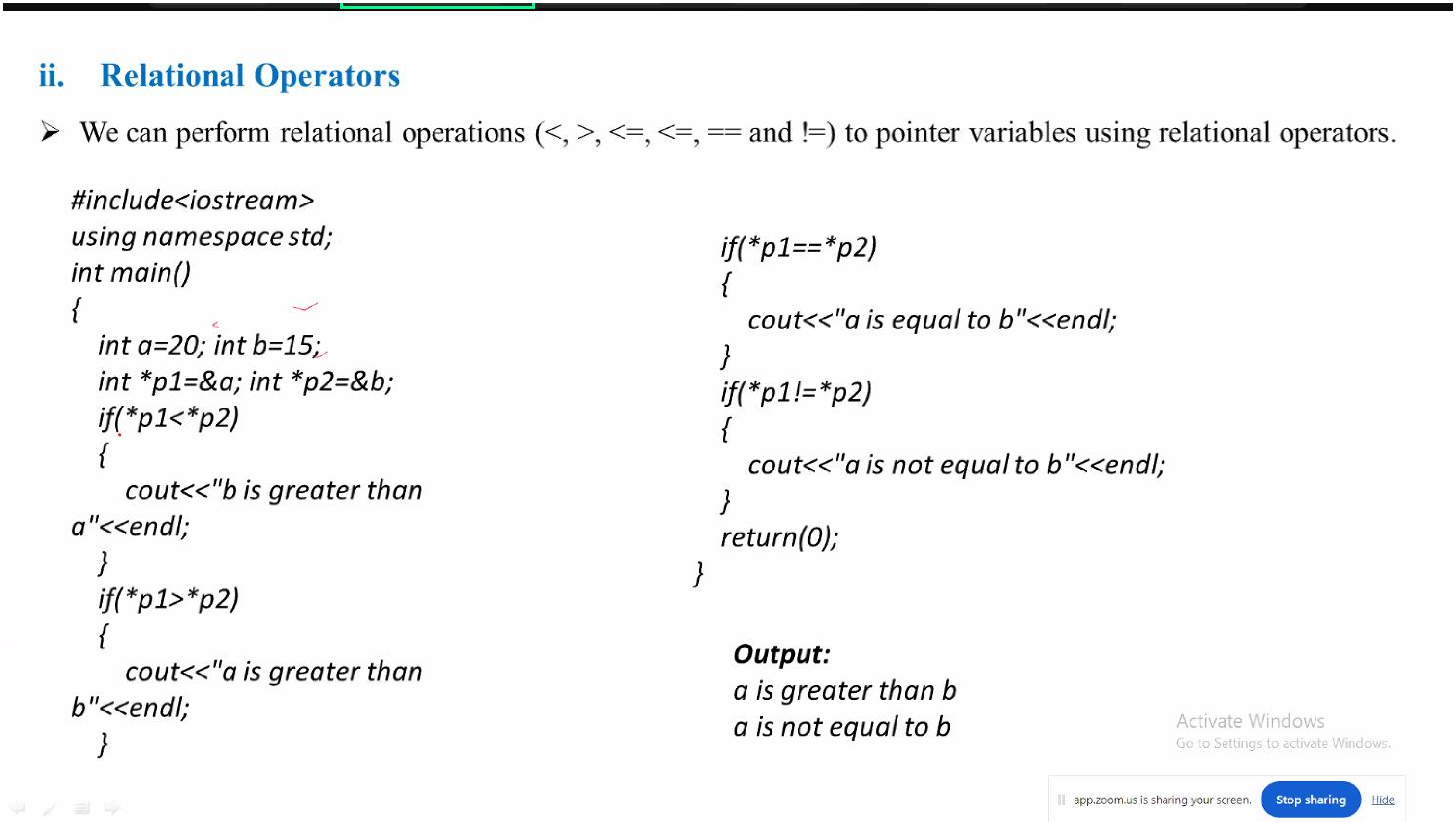
- We can perform arithmetic operations (+, -, *, / and %) to pointer variables using arithmetic operators.

```
#include<iostream>
using namespace std;
int main()
{
   int a=20; int b=15;
   int *p1=&a; int *p2=&b;
   cout<<*p1+*p2<<endl;
   cout<<*p1-*p2<<endl;
   cout<<*p1**p2<<endl;
   cout<<*p1/ *p2<<endl;
   cout<<*p1%*p2<<endl;
   cout<<*p1+*p2+2<<endl;
   return(0);
}
```

Output:

```
35
5
300
1
5
37
```

- A space should be there between / and *. /* is used for comment line.



ii. Relational Operators

- We can perform relational operations (<, >, <=, <=, == and !=) to pointer variables using relational operators.

```
#include<iostream>
using namespace std;
int main()
{
   int a=20; int b=15;
   int *p1=&a; int *p2=&b;
   if(*p1<*p2)
   {
      cout<<"b is greater than
a"<<endl;
   }
   if(*p1>*p2)
   {
      cout<<"a is greater than
b"<<endl;
   }
   if(*p1==*p2)
   {
      cout<<"a is equal to b"<<endl;
   }
   if(*p1!=*p2)
   {
      cout<<"a is not equal to b"<<endl;
   }
   return(0);
}
```

Output:

```
a is greater than b
a is not equal to b
```

iii. Assignment Operators

- We can perform assignment operations (=, +=, -=, *=, /= and %=) to pointer variables using assignment operators.

```
#include<iostream>
using namespace std;
int main()
{
   int a=20; int b=15;
   int *p1=&a; int *p2=&b;
   cout<<(*p1=11)<<endl;
   cout<<(*p1+=*p2)<<endl;
   cout<<(*p1-=*p2)<<endl;
   cout<<(*p1*=*p2)<<endl;
   cout<<(*p1/=*p2)<<endl;
   cout<<(*p1%=*p2)<<endl;
   return(0);
}
```

Output:

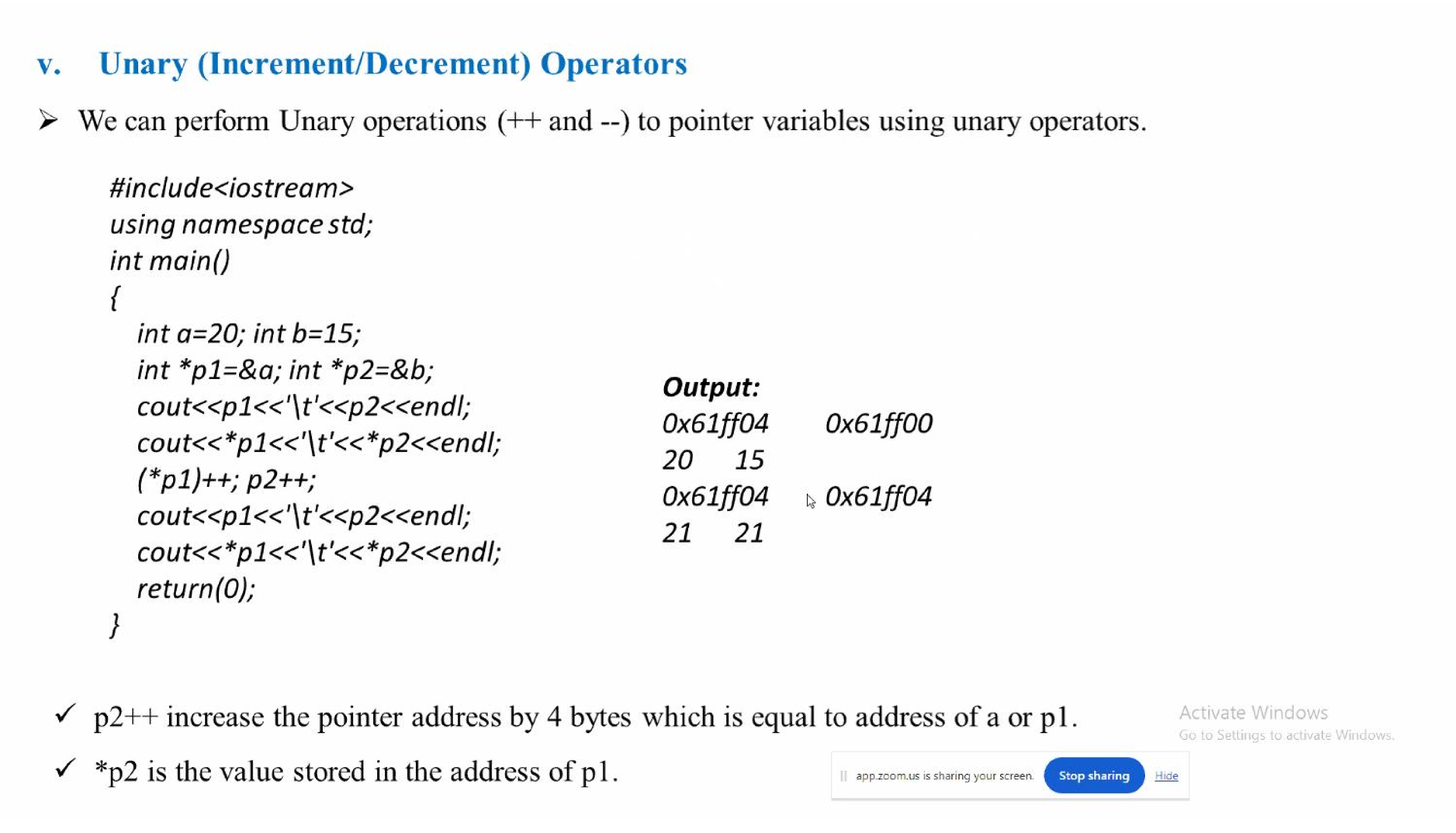
```
11
26
11
165
11
11
```

iv. Conditional Operators

- We can perform conditional operation to pointer variables using conditional operators.

```
#include<iostream>
using namespace std;
int main()
{
   int a=20; int b=15;
   int *p1=&a; int *p2=&b;
   (*p1>*p2)? cout<<"a is greater" :cout<<"b is greater";
   return(0);
}
```

Output: *a is greater*



v. Unary (Increment/Decrement) Operators

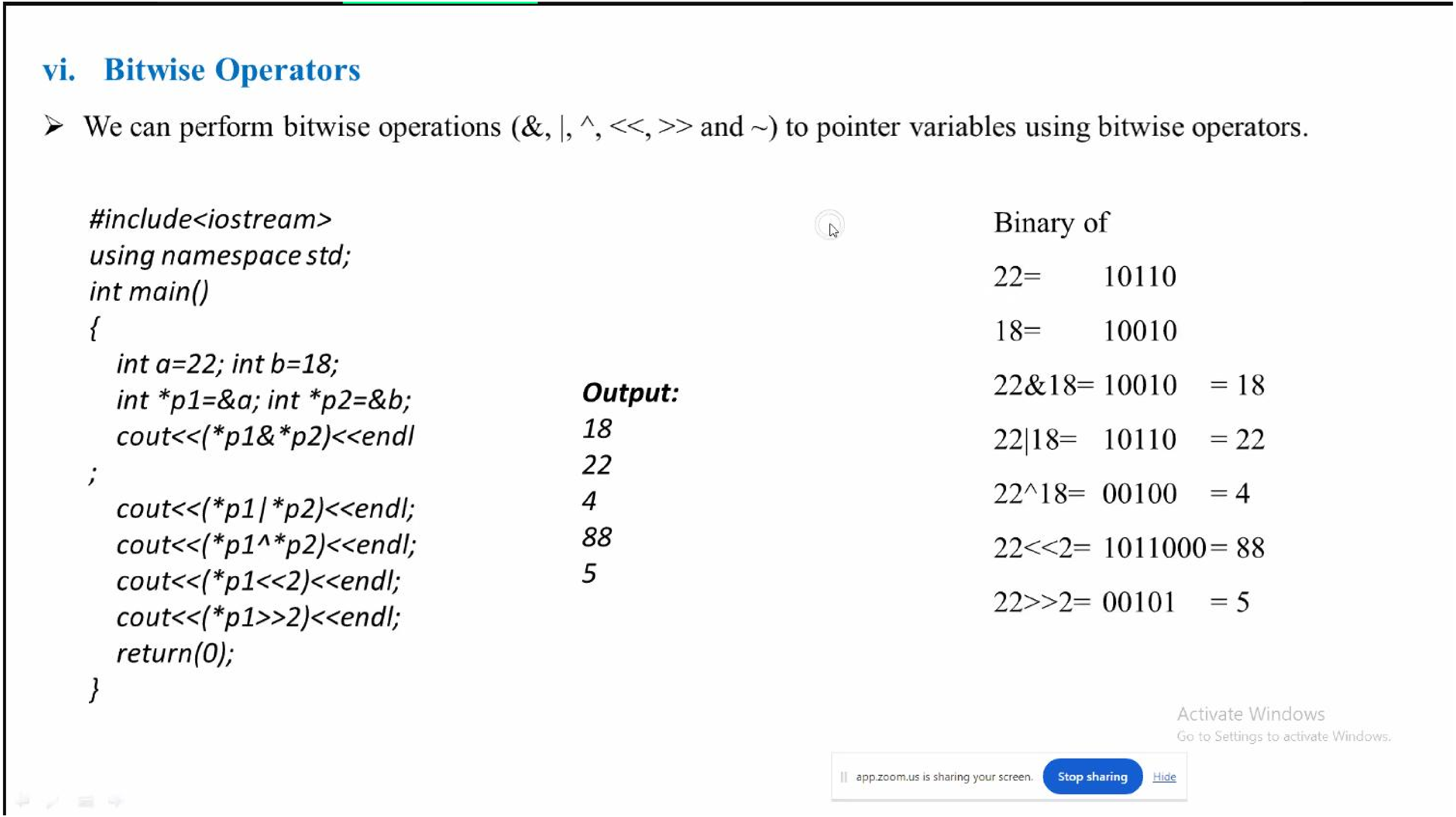
- We can perform Unary operations (++ and --) to pointer variables using unary operators.

```
#include<iostream>
using namespace std;
int main()
{
   int a=20; int b=15;
   int *p1=&a; int *p2=&b;
   cout<<p1<<'\t'<<p2<<endl;
   cout<<*p1<<'\t'<<*p2<<endl;
   (*p1)++; p2++;
   cout<<p1<<'\t'<<p2<<endl;
   cout<<*p1<<'\t'<<*p2<<endl;
   return(0);
}
```

Output:

```
0x61ff04      0x61ff00
20      15
0x61ff04      0x61ff04
21      21
```

- ✓ p2++ increase the pointer address by 4 bytes which is equal to address of a or p1.
- ✓ *p2 is the value stored in the address of p1.



vi. Bitwise Operators

- We can perform bitwise operations (&, |, ^, <<, >> and ~) to pointer variables using bitwise operators.

```
#include<iostream>
using namespace std;
int main()
{
   int a=22; int b=18;
   int *p1=&a; int *p2=&b;
   cout<<(*p1&*p2)<<endl
;
   cout<<(*p1|*p2)<<endl;
   cout<<(*p1^*p2)<<endl;
   cout<<(*p1<<2)<<endl;
   cout<<(*p1>>2)<<endl;
   return(0);
}
```

Output:

```
18
22
4
88
5
```

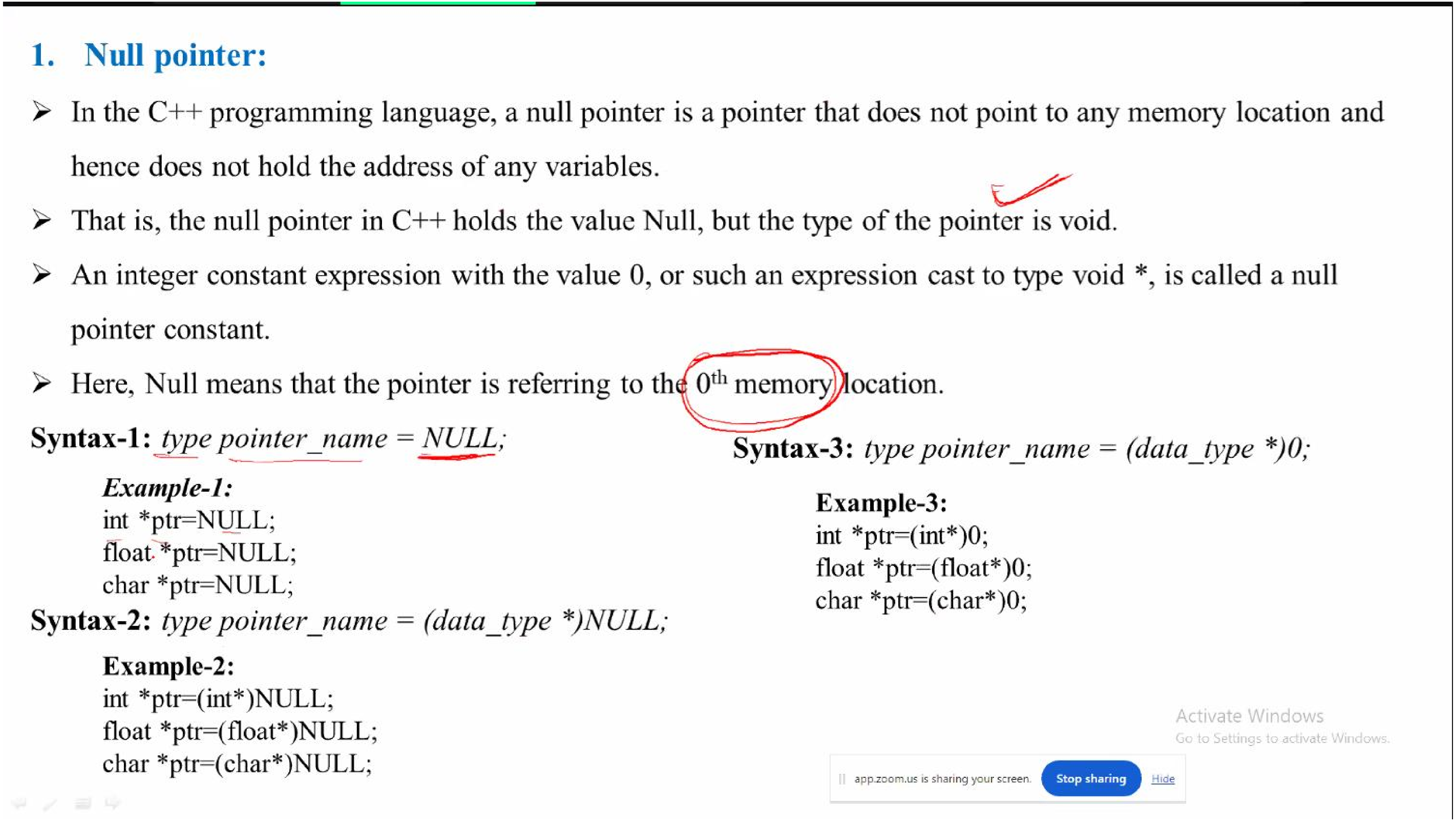
Binary of

22=	10110	
18=	10010	
22&18=	10010	= 18
22|18=	10110	= 22
22^18=	00100	= 4
22<<2=	1011000	= 88
22>>2=	00101	= 5

Types of Pointer:

There are eight different types of pointers which are as follows −

1. Null pointer
2. Void pointer
3. Wild pointer
4. Dangling pointer
5. Smart pointer
6. Complex pointer
7. Near pointer
8. Far pointer
9. Huge pointer



1. Null pointer:

- ➢ In the C++ programming language, a null pointer is a pointer that does not point to any memory location and hence does not hold the address of any variables.
- ➢ That is, the null pointer in C++ holds the value Null, but the type of the pointer is void.
- ➢ An integer constant expression with the value 0, or such an expression cast to type void *, is called a null pointer constant.
- ➢ Here, Null means that the pointer is referring to the 0th memory location.

Syntax-1: *type pointer_name = NULL;*

Example-1:

```
int *ptr=NULL;
float *ptr=NULL;
char *ptr=NULL;
```

Syntax-2: *type pointer_name = (data_type *)NULL;*

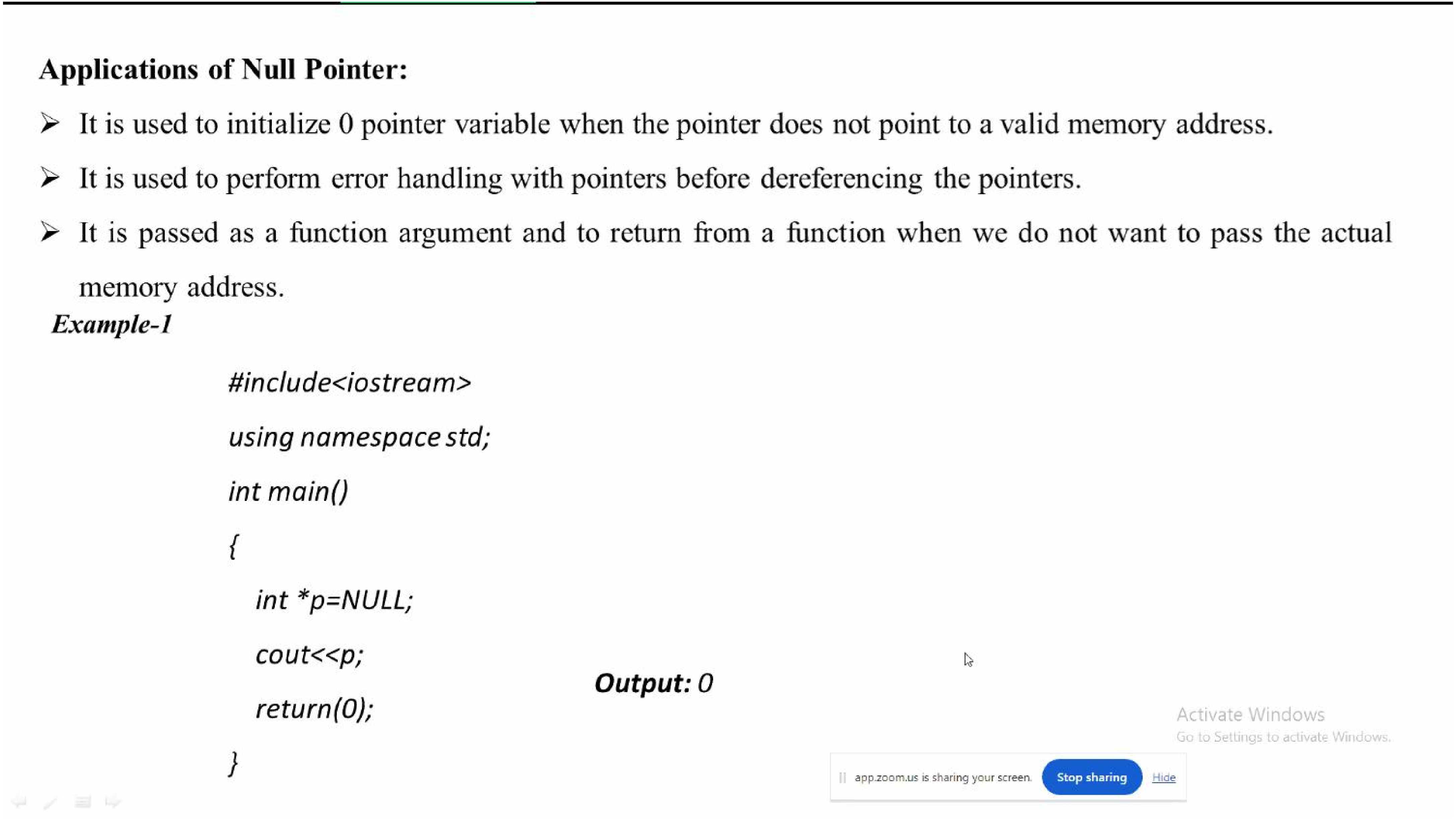
Example-2:

```
int *ptr=(int*)NULL;
float *ptr=(float*)NULL;
char *ptr=(char*)NULL;
```

Syntax-3: *type pointer_name = (data_type *)0;*

Example-3:

```
int *ptr=(int*)0;
float *ptr=(float*)0;
char *ptr=(char*)0;
```



Applications of Null Pointer:

- It is used to initialize 0 pointer variable when the pointer does not point to a valid memory address.
- It is used to perform error handling with pointers before dereferencing the pointers.
- It is passed as a function argument and to return from a function when we do not want to pass the actual memory address.

Example-1

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

Output: *0*