WHY POINTER SIZE WAS COMING 8 WHILE PRINTING?

In C++, the size of a pointer is architecture-dependent and can vary depending on the target platform. However, on most modern platforms, including those based on x86-64 architecture (which is the most common architecture for desktop and server systems), pointers are typically 8 bytes (64 bits) in size.

To understand why pointers are typically 8 bytes in size on modern platforms, we need to dive into the details of computer memory organization and the x86-64 architecture.

In computer memory organization, each byte of memory is assigned a unique address, and these addresses are used to access and manipulate data stored in memory. A pointer is a variable that stores the address of another variable in memory. When we declare a pointer in C++, the size of the pointer variable is determined by the size of the memory addresses used by the architecture.

The x86-64 architecture is a 64-bit extension of the x86 architecture, which has been used in Intel and AMD processors since the 1980s. One of the key features of the x86-64 architecture is that it supports a much larger address space than its 32-bit predecessor. Specifically, x86-64 supports a theoretical maximum of 2^64 bytes of addressable memory, which is far more than what is actually physically possible to implement in modern systems.

To enable this larger address space, x86-64 processors use 64-bit memory addresses, which require 8 bytes of storage. This means that any pointer variable in C++ that is used to store a memory address on an x86-64 platform must also be 8 bytes in size.

In addition to the memory address size, there are other factors that can affect the size of a pointer variable in C++. For example, some compilers may add padding or alignment bytes to ensure that the pointer is properly aligned in memory, which can increase its size. Additionally, some platforms may support different pointer sizes for different types of data (e.g., function pointers vs. data pointers).

Overall, the size of a pointer in C++ depends on a variety of factors, including the target platform's architecture, compiler implementation, and memory organization. However, on most modern platforms, including those based on x86-64 architecture, pointers are typically 8 bytes in size to support the larger address space enabled by 64-bit memory addresses.

WHY WE CANNOT DO [ARR = ARR + 1;] IN C++ ?

(AS DISCUSSED IN LAST CLASS)

In C++, the name of an array is actually a constant pointer to the first element of the array. This means that you cannot modify the value of the array pointer itself (i.e., the address of the first element of the array) by using pointer arithmetic.

So, if you try to do something like arr = 1, you will get a compilation error because you are trying to modify a constant pointer. This is because the pointer arr is pointing to the memory location of the first element of the array, and you cannot change this address.

However, you can use pointer arithmetic to access other elements of the array. For example, you can use arr + 1 to get a pointer to the second element of the array, like this:

```
int arr[5] = {1, 2, 3, 4, 5};

int *ptr = arr; // pointer to the first element of the array

ptr++; // ptr now points to the second element of the array

cout << *ptr << endl; // prints 2
```

In this example, ptr is initially pointing to the first element of the array arr. We then use the pointer arithmetic ptr++ to increment the pointer to point to the second element of the array. Finally, we dereference the pointer using *ptr to get the value of the second element of the array.

So, in summary, you cannot modify the value of the array pointer itself using pointer arithmetic, but you can use pointer arithmetic to access other elements of the array.