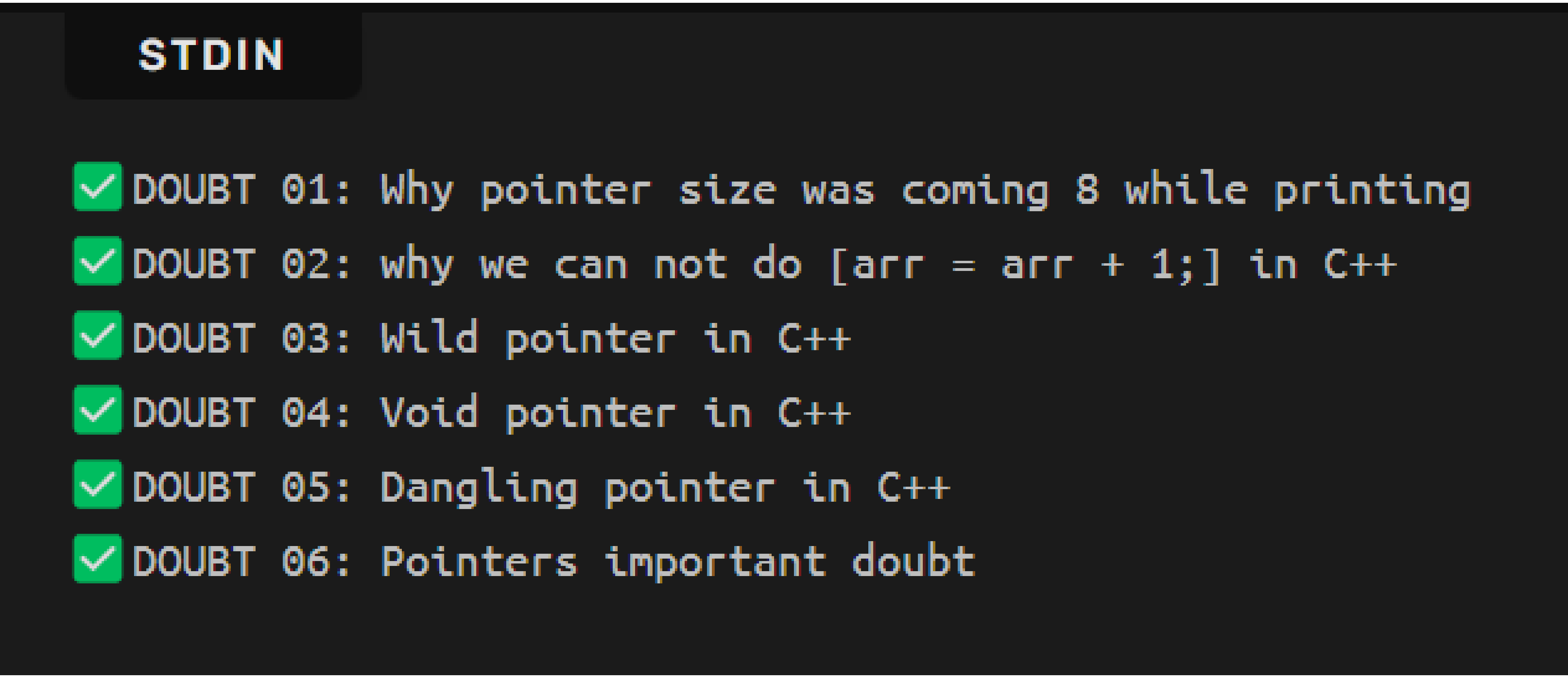


STDIN

✅ DOUBT 01: Why pointer size was coming 8 while printing

✅ DOUBT 02: why we can not do [arr = arr + 1;] in C++

✅ DOUBT 03: Wild pointer in C++

✅ DOUBT 04: Void pointer in C++

✅ DOUBT 05: Dangling pointer in C++

✅ DOUBT 06: Pointers important doubt

Why pointer size was coming 8 while printing ?

```
int a =5 ;
int* p = &a ;
```

↳ sizeof (P) = 8

```
char x = 'A' ;
char* p = &x ;
```

↳ sizeof (P) = 8

```
long y = 10 ;
long* y = & y ;
```

↳ sizeof (P) = 8

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.

Doubt:01 32-bit and 64-bit

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.

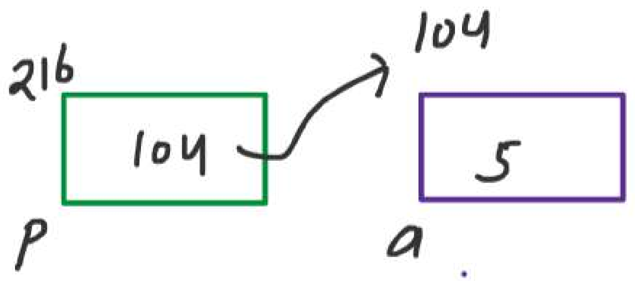
In addition to the memory address size, there are other factors that can affect the size of a pointer variable in C++. For example, some compilers may add padding or alignment bytes to ensure that the pointer is properly aligned in memory, which can increase its size. Additionally, some platforms may 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.

Note:02

(2)

```
int a = 5
int* p = &a
```



Print

```
p = p+1
cout << p ;
```

} output = Garbage value

```
int arr[5] = {10, 20, 30, 40, 50}
```

Print

```
arr = arr + 1 ;
cout << arr ;
```

} Compile Time ERROR

HW 02: why we can not do [arr = arr + 1;] in C++

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 = 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++; // pointer now points to the second element of the array

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

DOUBT-02

arr = arr + 1
↳ RUN TIME ERROR

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.

A wild pointer is a pointer in C++ that is uninitialized or has been deleted. This means that the pointer does not point to a valid memory location, and accessing or dereferencing a wild pointer can result in undefined behavior.

Here are some common scenarios where wild pointers can occur:

1. Uninitialized pointers If you declare a pointer variable without initializing it, the pointer will contain a random value that points to some memory location in the computer's memory. If you try to access or dereference this pointer, you may access memory that you should not access, which can cause your program to crash or behave unpredictably.

For example, consider the following code:

DOUBT-03
wild pointer

`int *ptr;`

`cout << *ptr << endl;`

In this example, `ptr` is declared but not initialized. When we try to dereference it using `*ptr`, we will get undefined behavior.

2. Deleting pointers If you delete a pointer and then try to access or dereference it, you will be accessing memory that has already been deallocated. This can cause your program to crash or behave unpredictably.

For example, consider the following code:

`int *ptr = new int;`

`delete ptr;`

`cout << *ptr << endl;`

In this example, we allocate memory for an integer using the `new` operator and assign the address of the allocated memory to `ptr`. We then delete the memory pointed to by `ptr` using the `delete` operator. When we try to access or dereference `ptr` using `*ptr`, we will get undefined behavior.

3. Pointers to non-existent variables If you create a pointer that points to a non-existent variable or object, you will be accessing memory that does not contain a valid object. This can cause your program to crash or behave unpredictably.

For example, consider the following code:

`int *ptr = &x;`

`cout << *ptr << endl;`

In this example, we create a pointer `ptr` that points to the address of an integer variable `x`. However, `x` has not been declared or initialized, so the pointer `ptr` points to a non-existent variable. When we try to access or dereference `ptr` using `*ptr`, we will get undefined behavior.

To avoid wild pointers in C++, you should always initialize your pointers to a valid memory location or to `nullptr` if they are not pointing to anything. You should also avoid deleting pointers that have not been allocated using the `new` operator. If you do delete a pointer, make sure to set it to `nullptr` to prevent it from becoming a wild pointer. Finally, you should always check if a pointer is pointing to a valid memory location before dereferencing it to avoid accessing a wild pointer.

A void pointer in C++ is a pointer that can point to any data type. Void pointers are also known as generic pointers because they can be used to point to any type of object or data. Void pointers are declared using the keyword `void` as the pointer type.

Here's an example of how to declare a void pointer:

```
void* ptr;
```

In this example, we declare a void pointer named `ptr`. This pointer can be used to point to any type of data, but we cannot dereference it directly since the compiler does not know the data type it is pointing to.

To use a void pointer in C++, we must first cast it to a specific data type. For example, if we want to use a void pointer to point to an integer, we can cast it to an `int*` data type:

Doubt : 04 void pointer

```
void* ptr;

int x = 10;

ptr = &x;

int* intPtr = static_cast<int*>(ptr);

cout << *intPtr << endl; // prints 10
```

In this example, we first declare a void pointer named `ptr` and an integer variable named `x`. We then assign the address of `x` to the void pointer `ptr`. We then cast the void pointer `ptr` to an `int*` pointer using the `static_cast` operator, which tells the compiler that the void pointer should be interpreted as a pointer to an integer. Finally, we dereference the integer pointer `intPtr` to get the value of `x`.

Void pointers are commonly used in C++ for dynamic memory allocation using the `new` operator. When we allocate memory dynamically using `new`, the type of the allocated memory is determined at runtime. We can use a void pointer to point to the dynamically allocated memory and then cast it to the appropriate data type when we need to use it.

Here's an example of using a void pointer for dynamic memory allocation:

`int* ptr = new int;`

`void* voidPtr = ptr;`

`int* intPtr = static_cast<int*>(voidPtr);`

`*intPtr = 10;`

`cout << *ptr << endl; // prints 10`

In this example, we allocate memory for an integer using the `new` operator and assign the address of the allocated memory to an integer pointer `ptr`. We then assign the void pointer `voidPtr` to `ptr`, which allows us to use the void pointer to point to the allocated memory. We then cast the void pointer `voidPtr` to an `int*` pointer using the `static_cast` operator and dereference it to set the value of the integer to 10. Finally, we print the value of the integer pointed to by `ptr` to confirm that it has been changed.

In summary, void pointers in C++ are pointers that can point to any type of data. They are commonly used for dynamic memory allocation and for passing pointers to functions that accept any type of data. However, since they cannot be dereferenced directly, we must first cast them to a specific data type before we can use them.

Meaning of static_cast<int*>:

In C++, `static_cast` is a casting operator that is used to convert a value from one data type to another. The syntax for using the static_cast operator is as follows:

`static_cast<new_type>(expression)`

Here, `new_type` is the data type that we want to cast the `expression` to. The expression can be a variable, a literal, or any valid expression in C++.

In the context of `static_cast<int*>`, the `int*` is the new data type that we want to cast the expression to. Specifically, it is a pointer to an integer. In other words, we are casting a void pointer to an integer pointer.

When we cast a void pointer to a specific data type using static_cast, we are telling the compiler that we want to treat the memory location that the void pointer points to as a pointer to the new data type. This allows us to dereference the pointer and access the value of the new data type.

For example, suppose we have a void pointer `voidPtr` that points to an integer. We can cast it to an integer pointer using static_cast as follows:

`void* voidPtr = // some memory location that stores an integer`

`int* intPtr = static_cast<int*>(voidPtr);`

Now, we can use the integer pointer `intPtr` to access the integer value that the void pointer `voidPtr` points to.

Note that static_cast is a compile-time operator, which means that the conversion is done at compile time rather than at runtime. This can result in faster code execution since the compiler can optimize the code based on the specific data types that are being used. However, it also means that the cast can fail at runtime if the data types are not compatible.

Overall, static_cast<int*> is a C++ casting operator that is used to cast a void pointer to an integer pointer.

Dangling pointers are a common problem in C++ programming that can lead to unpredictable behavior and even program crashes. In this article, we'll explore what dangling pointers are, how they arise, and some strategies for avoiding them.

What is a Dangling Pointer?

A dangling pointer is a pointer that points to a memory location that has been freed or deallocated. When you dereference a dangling pointer, the program may crash or exhibit undefined behavior. This can be a serious problem in C++, where pointers are widely used to manage memory and data structures.

Dangling pointers typically arise when you delete or free a block of memory that a pointer points to, but then fail to update the pointer to reflect the deallocation. The pointer still points to the original memory location, even though that memory has been freed and may have been reused for other purposes.

For example, consider the following code:

Doubt : 5 DANGLing Pointer

```
int* p = new int;

*p = 42;

delete p;

cout << *p << endl; // Undefined behavior
```

In this code, we allocate a new integer on the heap and assign the value 42 to it. We then delete the pointer `p`, which frees the memory block that `p` points to. However, we then try to dereference `p` and print its value to the console, which is undefined behavior since `p` now points to a freed memory location.

How Do Dangling Pointers Arise?

Dangling pointers typically arise when you manipulate memory using pointers and fail to properly manage the lifetime of that memory. Here are some common scenarios where dangling pointers can occur:

1.
Freeing memory while a pointer still points to it: This is the most common scenario for creating a dangling pointer. If you delete or free memory while a pointer still points to it, the pointer becomes a dangling pointer.

2.
Returning a pointer to a local variable: If you return a pointer to a local variable from a function, the pointer becomes a dangling pointer as soon as the function returns and the local variable goes out of scope.

3.
Using a pointer after it has been deleted or freed: If you continue to use a pointer after you have deleted or freed the memory it points to, the pointer becomes a dangling pointer.

How to Avoid Dangling Pointers?

The best way to avoid dangling pointers is to use smart pointers or RAII (Resource Acquisition Is Initialization) techniques. These techniques ensure that memory is automatically managed and freed when it is no longer needed, eliminating the need for explicit memory management.

Here are some strategies for avoiding dangling pointers in your code:

1. Use smart pointers: C++ provides several types of smart pointers, such as unique_ptr, shared_ptr, and weak_ptr, that automatically manage memory and prevent dangling pointers. Smart pointers use RAII techniques to ensure that memory is freed when it is no longer needed.

2. Use references instead of pointers: If you don't need to modify the value pointed to by a pointer, you can use a reference instead. References are similar to pointers but do not require explicit memory management.

3. Avoid returning pointers to local variables: If you need to return a value from a function, consider using a reference or a smart pointer instead of a raw pointer. If you must use a raw pointer, make sure the memory it points to is not freed before the pointer is used.

4. Nullify pointers after deleting or freeing memory: To prevent dangling pointers, always nullify a pointer after deleting or freeing the memory it points to. This ensures that the pointer is no longer a dangling pointer and can be safely used without causing undefined behavior.

In summary, dangling pointers are a common problem in C++ programming that can lead to unpredictable behavior and program crashes. To avoid dangling pointers, use smart pointers.

In C++, an array of pointers is an array where each element is a pointer to a memory location, while a pointer to an array is a pointer that points to the first element of an array.

For example, consider the following code:

Pointer Imp
Doubts
↳ 6

```
int* arr[3]; // array of pointers to int

int nums[3] = {1, 2, 3};

int* ptr = nums; // pointer to int (points to the first element of nums)

// assigning pointers to elements in the array of pointers

arr[0] = &nums[0];

arr[1] = &nums[1];

arr[2] = &nums[2]; // accessing elements using array notation

cout << *arr[0] << endl; // prints 1

cout << *arr[1] << endl; // prints 2

cout << *arr[2] << endl; // prints 3

// accessing elements using pointer notation

cout << *ptr << endl; // prints 1

cout << *(ptr + 1) << endl; // prints 2
cout << *(ptr + 2) << endl; // prints 3
```

In this example, `arr` is an array of pointers to `int`, and `ptr` is a pointer to the first element of the `nums` array. The `arr` array is initialized to point to the elements of the `nums` array. We can access the elements of the `arr` array using array notation, or we can access the elements of the `nums` array using pointer notation.

In summary, an array of pointers is an array where each element is a pointer, while a pointer to an array is a pointer that points to the first element of an array.

`int (*ptr)[10]` is a pointer to an array of 10 integers in C++. This means that `ptr` is a pointer that points to the first element of an array that contains 10 integers.

For example, consider the following code:

```
int nums[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

int (*ptr)[10] = &nums; // pointer to an array of 10 integers

// accessing elements of the array using pointer notation

cout << (*ptr)[0] << endl; // prints 1

cout << (*ptr)[1] << endl; // prints 2

cout << (*ptr)[2] << endl; // prints 3
```

In this example, `nums` is an array of 10 integers, and `ptr` is a pointer to the first element of `nums`. The parentheses around `*ptr` are necessary because the dereference operator `*` has a lower precedence than the array subscript operator `[]`.

We can access the elements of the array using pointer notation, but we need to use the parentheses to first dereference the pointer and then apply the subscript operator to the array.

In summary, `int (*ptr)[10]` is a pointer to an array of 10 integers in C++.