README

What are pointers

In C++, a pointer refers to a variable that holds the address of another variable. Like regular variables, pointers have a data type.

For example, a pointer of type integer can hold the address of a variable of type integer. A pointer of character type can hold the address of a variable of character type.

Addresses in C++

&

For example, if x is a variable, &x returns the address of the variable.

Pointer Declaration Syntax

The declaration of C++ takes the following syntax:

datatype *variable_name;

- The datatype is the base type of the pointer which must be a valid C++ data type.
- The variable_name is should be the name of the pointer variable.
- Asterisk used above for pointer declaration is similar to asterisk used to perform multiplication operation. It is the asterisk that marks the variable as a pointer.

Here is an example of valid pointer declarations in C++:

```cpp
int    *x;     // a pointer to integer
double *x;     // a pointer to double
float  *x;     // a pointer to float
char   *ch      // a pointer to a character
```

Reference operator (&) and Deference operator (`*`)

The reference operator (&) returns the variable's address.

The dereference operator (`*`) helps us get the value that has been stored in a memory address.

For example:

If we have a variable given the name num, stored in the address 0×234 and storing the value 28.

The reference operator (&) will return 0×234.

The dereference operator (`*`) will return 5.

(ampersand) & : Address of
(asterisk) `*`: Value of / Pointer to

Here is how we can declare pointers.

```cpp
int *pointVar;
```

Here, we have declared a pointer pointVar of the `int` type.

We can also declare pointers in the following way.

```cpp
int* pointVar; // preferred syntax
```

Let's take another example of declaring pointers.

```cpp
int* pointVar, p;
```

Assigning Addresses to Pointers

Here is how we can assign addresses to pointers:

```cpp
int* pointVar, var;
var = 5;

// assign address of var to pointVar pointer
pointVar = &var;
```

V/s

```cpp
int* pointVar = &var;
```

V/s

language-cpp

```cpp
int *pointVar;
*pointVar = &var;
```

Here, `5` is assigned to the variable var. And, the address of var is assigned to the pointVar pointer with the code `pointVar = &var`.

Get the Value from the Address Using Pointers

To get the value pointed by a pointer, we use the `*` operator. For example:

language-cpp

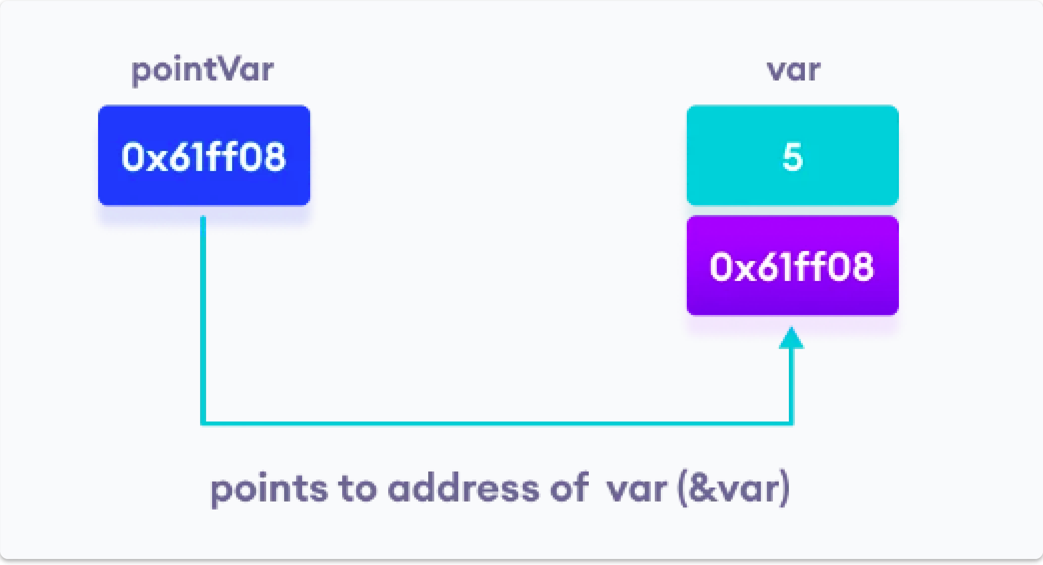
```cpp
int* pointVar, var;
var = 5;

// assign address of var to pointVar
pointVar = &var;

// access value pointed by pointVar
cout << *pointVar << endl;   // Output: 5
```

In the above code, the address of var is assigned to pointVar. We have used the `*pointVar` to get the value stored in that address.

When `*` is used with pointers, it's called the dereference operator. It operates on a pointer and gives the value pointed by the address stored in the pointer. That is, `*pointVar = var`.



Changing Value Pointed by Pointers

If pointVar points to the address of var, we can change the value of var by using *pointVar.

For example,

```cpp
int var = 5;
int* pointVar;

// assign address of var
pointVar = &var;

// change value at address pointVar
*pointVar = 1;

cout << var << endl; // Output: 1
```

Here, pointVar and `&var` have the same address, the value of var will also be changed when *pointVar is changed.

Common mistakes when working with pointers

Suppose, we want a pointer varPoint to point to the address of var. Then,

```cpp
int var, *varPoint;

// Wrong!
// varPoint is an address but var is not
varPoint = var;

// Wrong!
// &var is an address
// *varPoint is the value stored in &var
*varPoint = &var;

// Correct!
// varPoint is an address and so is &var
varPoint = &var;

 // Correct!
// both *varPoint and var are values
*varPoint = var;
```

Pointers and Arrays

Arrays and pointers work based on a related concept. There are different things to note when working with arrays having pointers. The array name itself denotes the base address of the array. This means that to assign the address of an array to a pointer, you should not use an ampersand (&).

For example:

```cpp
p = arr;
```

language-cpp

The above is correct since `arr` represents the arrays' address. Here is another example:

X

```cpp
p = &arr;
```

language-cpp

The above is incorrect.

We can implicitly convert an array into a pointer. For example:

```cpp
int arr [20];
int * ip;
```

language-cpp

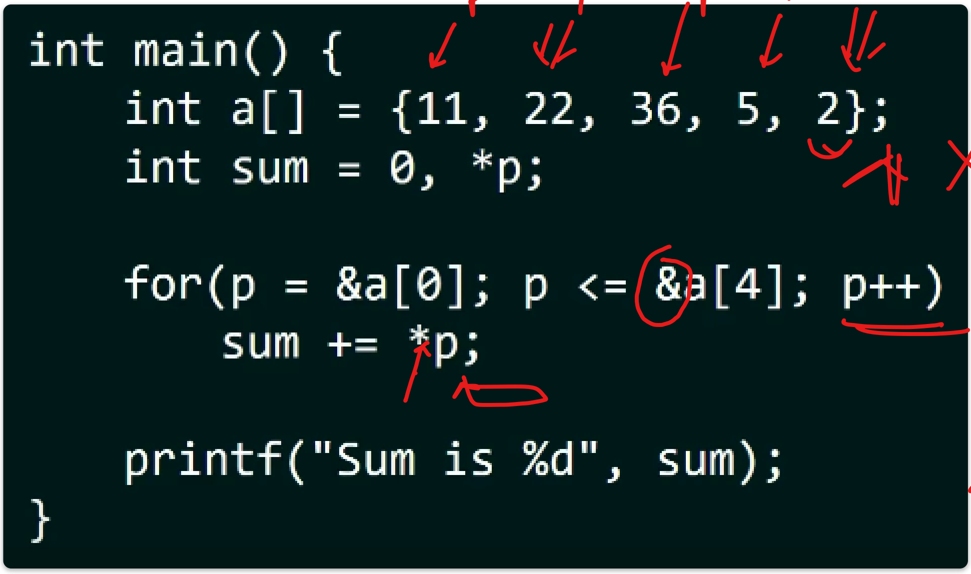
Below is a valid operation:

✓

```cpp
ip = arr;
```

language-cpp

After the above declaration, ip and `arr` will be equivalent, and they will share properties. However, a different address can be assigned to ip, but we cannot assign anything to `arr`.



```c
int main() {
    int a[] = {11, 22, 36, 5, 2};
    int sum = 0, *p;

    for(p = &a[0]; p <= &a[4]; p++)
        sum += *p;

    printf("Sum is %d", sum);
}
```

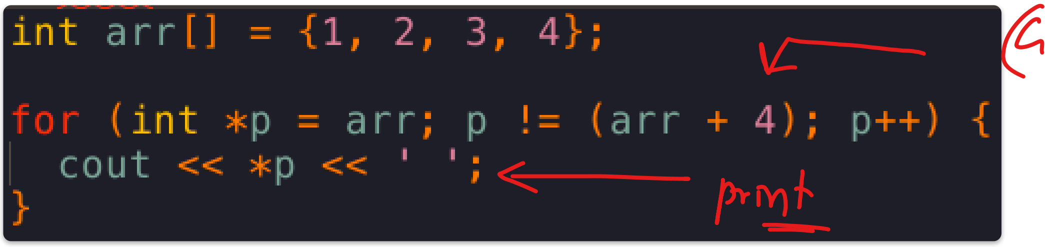
p p p p p

X

sumre value of (p)

p ⇐ arr[0];

int size= 4



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

for (int *p = arr; p != (arr + 4); p++) {
  cout << *p << ' ';
}
```

(arr + size)

print

nullptr

int *ptr;

*ptr = 500

C++

int *ptr= nullptr;

NULL

C

void pointer

This is a special type of pointer available in C++ which represents the absence of type. Void pointers are pointers that point to a value that has no type (and thus also an undetermined length and undetermined dereferencing properties). This means that void pointers have great flexibility as they can point to any data type.

void *ptr;

ptr

There is a payoff for this flexibility. These pointers cannot be directly dereferenced. They have to be first transformed into some other pointer type that points to a concrete data type before being dereferenced.

Questions

1. two

```cpp
#include <iostream>
using namespace std;
int main() {
    int num[5];
    int* p;

    p = num;

    *p = 10;
    p++;

    *p = 20;
    p = &num[2];

    *p = 30;
    p = num + 3;

    *p = 40;
    p = num;
```

```
        *(p + 4) = 50;

    for (int i = 0; i < 5; i++)
        cout << num[i] << " ";

    return 0;
}
```

▶ solution

2.

```cpp
#include <iostream>
using namespace std;
int main()
{
    int arr[] = { 4, 5, 6, 7 };
    int* p = (arr + 1);
    cout << *arr + 10;
    return 0;
}
```

p

4567 15

4

* → value of (arr)

arr gives you address of arr[5];

▶ solution

New operator | delete

In C++, when you declare a pointer variable using the `new` operator, as in the following code:

To declare an array using a pointer, you first declare a pointer variable and allocate memory for the array using the `new` operator. The `new` operator returns a pointer to the first element of the array, which you can assign to the pointer variable

```cpp
int* arr = new int[5];
```

the memory for the array is allocated dynamically on the heap, and the address of the first element of the array is returned by the `new` operator.

The pointer variable `arr` itself is stored on the stack, which is a region of memory allocated to the current function call. The memory allocated for the array elements is separate from the memory allocated for the pointer variable itself.

When you are done with the array, you should deallocate the memory on the heap using the `delete[]` operator, like this:

```cpp
delete[] arr;
```

This frees the memory previously allocated for the array, preventing memory leaks.

It is worth noting that when you declare an array using the syntax `int arr[5]`, as opposed to using the `new` operator, the memory for the array is allocated on the stack, and there is no need to use the `delete` operator to deallocate the memory. However, the size of the array must be known at compile time, whereas with dynamic memory allocation using `new`, the size of the array can be determined at runtime.