C++ Pointers

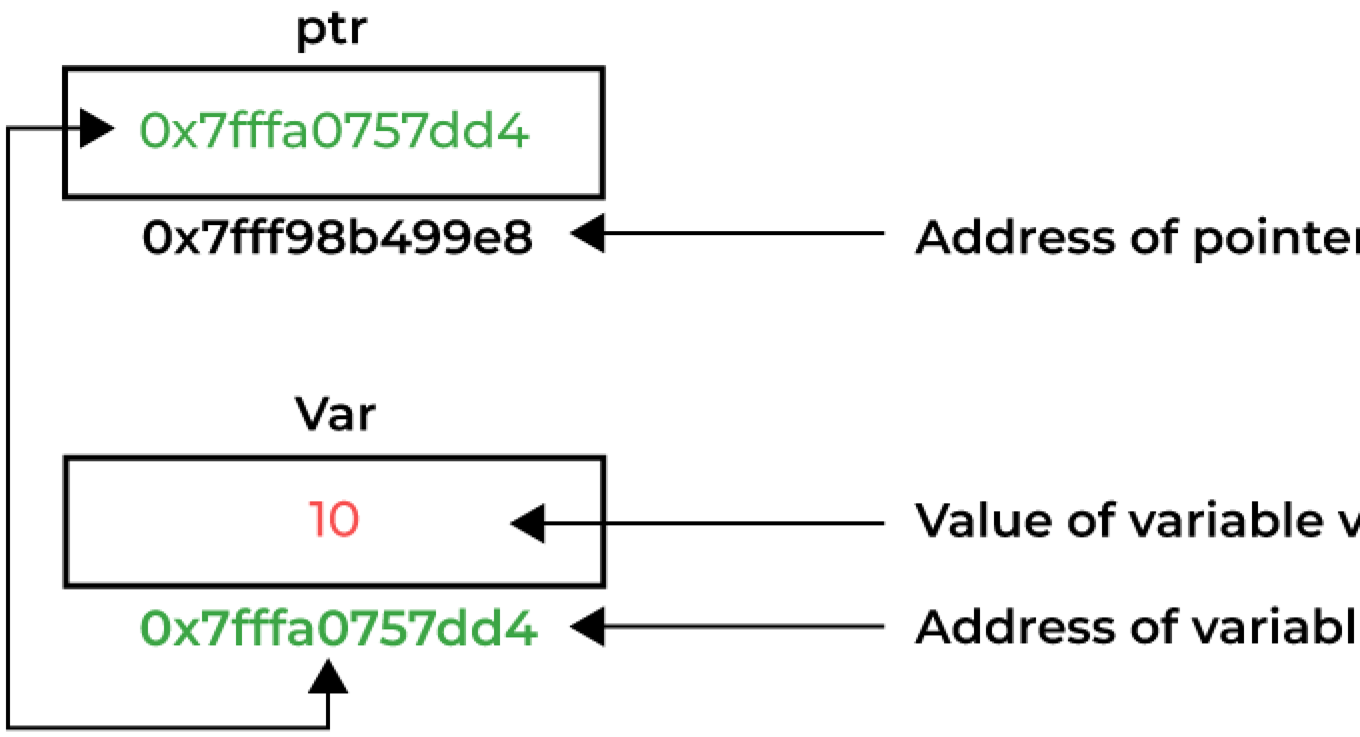
A pointer is a special variable that holds the memory address of another variable, rather than storing a direct value itself. Pointers allow programs to access and manipulate data in memory efficiently, making them a key feature for system-level programming and dynamic memory management. When we access a pointer directly, we get the address it holds not the actual data stored at that location.

```cpp
#include <iostream>
using namespace std;

int main() {
    int var = 10;

    // declare pointer and store address of x
    int* ptr = &var;

    // print value and address
    cout << "Value of x: " << var << endl;
    cout << "Address of x: " << &var << endl;
    cout << "Value stored in pointer ptr: " << ptr << endl;
    cout << "Value pointed to by ptr: " << *ptr << endl;

    return 0;
}
```

Output

```
Value of x: 10
Address of x: 0x7fffa0757dd4
Value stored in pointer ptr: 0x7fffa0757dd4
Value pointed to by ptr: 10
```

In the code above:



- **int* ptr;** declares a pointer to an integer.
- **ptr = &x;** stores the address of variable x in the pointer ptr.
- ***ptr** is called the dereference operator. It gives us access to the value stored at the memory address ptr is pointing to.

So, ptr holds the address of x, and *ptr gives the value of x by accessing that address.

Create Pointer

A pointer can be declared in the same way as any other variable but with an **asterisk symbol (*)** as shown:

```
data_type* name
```

Here, **data_type** is the type of data that the pointer is pointing to, and **name** is the name assigned to the pointer. The *** symbol** is also called **dereference operator.**

Example:

```
int* ptr;
```

In the above statement, we create a pointer **ptr** that can store the address of an integer data. It is pronounced as **"Pointer to Integer"** or **"Integer Pointer"**

Assign Address

The **addressof operator (&)** determines the address of any variable in C++. This address can be assigned to the pointer variable to initialize it.

Example:

```
int val = 22;
int* ptr = &val;
```

In the above statement, pointer **ptr** store the address of variable **val** using address-of operator **(&).** The pointer and the variable should be of same type, Otherwise type mismatch error occurs.

Dereferencing

The process of accessing the value present at the memory address pointed by the pointer is called dereferencing. This is done with the help of dereferencing operator as shown:

```
#include <iostream>
using namespace std;

int main() {
    int var = 10;

    // Store the address of
    // var variable
    int* ptr = &var;

    // Access value using (*)
    // operator
    cout << *ptr;
    return 0;
}
```

Output

```
10
```

Directly accessing the pointer will just give us the address that is stored in the pointer.

```
#include <iostream>
using namespace std;

int main() {
    int var = 10;
    // Store the address of
    // var variable
    int* ptr = &var;

    // Access the address value
    cout << ptr;
    return 0;
}
```

Output

```
0x7fffa0757dd4
```

Modify Address

The pointer can be modified to point to another memory address if its is of the same type.

Example:

```
#include <iostream>
using namespace std;

int main() {
    int a = 10;
    int b = 99;

    int *ptr = &a;
    cout << *ptr << endl;

    // Changing the address stored
    ptr = &b;
    cout << *ptr;

    return 0;
}
```

Output

```
10

99
```

Size of Pointers

The size of pointer in a system is equal for every pointer no matter what type of data it is pointing to. It does not depend on the type, but on operating system and CPU architecture. The size of pointers in C++ is

- **8 bytes** for a **64-bit System**
- **4 bytes** for a **32-bit System**

The logic is simple: pointers store the addresses of the memory and in a computer, the maximum width of a memory address is determined by the CPU architecture. For example, for a 64-bit CPU, the address will always be 64-bit wide. This can be verified using sizeof operator.

```
#include <iostream>
using namespace std;

int main() {
    int *ptr1;
    char *ptr2;

    // Finding size using sizeof()
    cout << sizeof(ptr1) << endl;
    cout << sizeof(ptr2);

    return 0;
}
```

Output

```
8

8
```

As we can see, both the pointers have same size. It's a 64-bit system, so the size is 8 bytes.

Special Types of Pointers

There are 4 special types of pointers that used or referred to in different contexts:

Wild Pointer

When a pointer is created, it just contains a random address that may or may not be valid. This type of pointer is called **wild pointer**.

```
#include <iostream>
using namespace std;

int main() {
    // Wild pointer
    int *ptr;
    return 0;
}
```
Dereferencing this pointer may lead to errors such as segmentation faults. So, it is always recommended to initialize a pointer.

NULL Pointer

A **NULL pointer** is a pointer that does not point to any valid memory location but NULL. It is often used to initialize a pointer when you do not want it to point to any object.
```
#include <iostream>
using namespace std;

int main() {

    // NULL pointer
    int *ptr = NULL;

    return 0;
}
```

Void Pointers

A **void pointer** (void*) is a special type of pointer in C++ that has no associated data type. It can hold the address of any data type, making it useful for generic programming. However, since the type is unknown, the compiler doesn't know how many bytes to read or how to interpret the data. Therefore, **a void pointer cannot be directly dereferenced**. It must first be explicitly typecast to the appropriate pointer type.
```
#include <iostream>
using namespace std;

int main() {
    int x = 42;

    // void pointer holding address of an int
    void* ptr = &x;

    // Error: cannot dereference void pointer
    // cout << *ptr;

    // Typecast before dereferencing
    cout << "Value pointed by void pointer: " <<
*(static_cast<int*>(ptr)) << endl;

    return 0;
}
```

Output

```
Value pointed by void pointer: 42
```

Dangling Pointer

A **dangling pointer** is a pointer that refers to memory which has already been freed or is no longer valid. This typically happens when:

- A pointer points to a local variable that has gone out of scope
- Dynamically allocated memory is deallocated using delete, but the pointer still holds the old address

Dereferencing a dangling pointer leads to **undefined behavior**, and is a common source of hard-to-find bugs.

```
#include <iostream>
using namespace std;

int* getPointer() {
    int x = 10;

    // returning address of local variable
    return &x;
}

int main() {

    // ptr becomes dangling here
    int* ptr = getPointer();

    // Undefined behavior
    // cout << *ptr;
    return 0;
}
```

Pointer Arithmetic

Pointer arithmetic refers to the operations that C++ allows on the pointers. They include:

- **Incrementing and Decrementing**
- **Addition of Integer**
- **Subtraction of Integer**
- **Subtraction of Two Pointers of the Same Type**
- **Comparison of Pointers and NULL**

Pointer To Pointer (Double Pointer)

We can also have pointers that point to other pointers. This is called a double pointer, and it is useful when working with multi-level data structures like arrays of pointers or dynamic memory management.

Example:

```
#include <iostream>
using namespace std;

int main() {
    int var = 10;

    // Store the address of
    // var variable
    int* ptr1 = &var;
    int** ptr2 = &ptr1;

    // Access values using (*)
    // operator
    cout << *ptr1 << endl;
    cout << **ptr2;
    return 0;
}
```

Output

```
10

10
```

Pointer to Functions

In C++, a function pointer is used to point to functions, similar to how pointers are used to point to variables. It allows you to save the address of a function. Function pointers can be used to call a function indirectly, or they can be passed as arguments to another function, enabling dynamic function invocation and flexibility in function handling.

Smart Pointers

A smart pointer is a wrapper class around a pointer that overloads operators like * and ->. Smart pointer objects resemble normal pointers, they have the added functionality of automatically deallocating and freeing the memory of the object when it is no longer needed, unlike regular pointers.

Pointer vs Reference

Understanding the differences between pointers and references in C++. Both are used to refer to other variables, but they operate in different ways and have different behavior.

Aspect	Pointer	Reference
Initialization	A pointer can be initialized after declaration.	A reference must be initialized at the time of declaration.
Nullability	A pointer can be assigned NULL or nullptr.	A reference cannot be null, it must always refer to a valid object.
Reassignment	A pointer can be reassigned to point to different objects.	A reference cannot be reassigned once it is bound to an object.

Uses of Pointers

Pointers are a useful concept in C++ that allow direct access to memory addresses, providing greater control over memory and data manipulation. Below are some primary uses of pointers in C++:

- **Dynamic Memory Allocation:** Pointers allow memory to be allocated dynamically at runtime using operators like new and delete. This enables the creation of objects or arrays whose sizes are determined during execution.

- **Implementing Data Structures:** Pointers are used to implementing complex data structures such as linked lists, trees, and graphs, where elements are dynamically allocated and linked together.
- **Pass Arguments by Pointer:** Pass the argument with their address and make changes in their value using pointer. So that the values get changed into the original argument.