Discussion-3 C++ Pointers

<u>Discussion Topic:</u>

In this module, we will explore working with pointers in the C++ programming language.

- How are pointers used with arrays, and what are the benefits to utilizing pointers?
- Are arrays necessary when using a pointer?
- Additionally, what is the difference between the address-of operator and the dereference operator in C++?

Be sure to provide an appropriate example to illustrate your viewpoints.

<u>My Post:</u>

Hello Class,

The C++ programming language provides memory management through pointers. Pointers offer flexibility and precise control over memory, making them a powerful tool for applications where memory usage management and manipulation are needed. This post discusses how pointers interact with arrays, whether arrays are necessary when using pointers, how pointer arithmetic relates to arrays and the differences between the address-of operator (`&`) and the dereference operator (`*`). Additionally, the post refers to Unique pointers.

Pointers and Arrays: How They Work Together

In C++, pointers and arrays work well together, but arrays are not necessarily required when using pointers and vice versa. However, they are often used together due to their similar nature, as they use direct memory access; that is arrays use indexing to access elements, while pointers use memory addresses to access elements. Both are derived data types in C++ that have a lot in common (Rinkalktailor, 2024); a derived data type is a data type that is created or derived from the fundamental or built-in data types. In some cases, pointers can even replace arrays. However, if they are very closely related they are still different data types. Below is a table listing the similarities and differences between the two data types.

Table 1

Arrays vs Pointers

Array	**Pointer**
Arrays are declared as type var_name[size];	Pointers are declared as type * var_name;
Collection of elements of similar data type.	Store the address of another variable.
The array can be initialized at the time of definition.	Pointers can also be initialized at definition.
The size of the array decides the number of elements it can store.	The pointer can store the address of only one variable.
Arrays are allocated at compile time.	Pointers are allocated at run-time.
Memory allocation is contiguous.	Memory allocation is random.
Arrays are static in nature i.e. they cannot be resized according to the user requirements.	Pointers are dynamic in nature i.e. memory allocated can be resized later.
An array of pointers can be created.	A pointer to an array can be created.

Note: From “Pointers vs Array in C++” by Rinkalktailor (2024), modify.

As mentioned earlier arrays are not strictly necessary when using pointers, and vice versa. Below are three examples that illustrate the interaction between arrays and pointers, pointer arithmetic, and how pointers can be used to access array elements, as well as an example of how to use a pointer as an array.

Example 1: Accessing Array Elements Using Indexing

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

for (int i = 0; i < 5; ++i) {
    std::cout << arr[i] << " ";  // Uses a variable as an index to access element in
                                 // the array
} // Outputs 1 2 3 4 5
```

This is the typical method used to access elements in arrays.

Example 2: Accessing Array Elements Using Pointer Arithmetic

```
int arr[5] = {1, 2, 3, 4, 5};
int *ptr = arr;  // arr acts like a pointer to the first element

for (int i = 0; i < 5; ++i) {
    std::cout << *(ptr + i) << " ";  // Dereferencing the pointer to access array
                                     // elements
} // Outputs 1 2 3 4 5
```

This method uses pointer arithmetic '*(*(ptr + i))*' to access the element in the array. Pointer arithmetic allows incrementing (++) and decrementing (--) pointers, or adding/subtracting integers to pointers to access elements in the array (Yashk, 2023). See this post section on Pointer Arithmetic for more information. In essence, in this example, the pointer is used as an array.

Example 3: Pointer Arithmetic to Traverse an Array

```
int arr[5] = {1, 2, 3, 4, 5};
int *ptr = arr;
std::cout << *ptr << " ";    // Outputs 1
ptr++;
std::cout << *ptr << " ";    // Outputs 2
ptr += 2;
std::cout << *ptr << " ";    // Outputs 4
```

This example is similar to the previous one, it also used the pointer is used as an array and pointer arithmetic. Note that, the code line '*int *ptr = arr;*' the pointer '*ptr*' points to the first element of the array. In C++, when assigning an array to a pointer, the pointer points to the first element of that array.

Pointers and arrays in C++ work well together. While similar, they diverge in how they handle memory allocation and how elements are accessed. Additionally, pointers can be directly manipulated using pointer arithmetic.

Pointer Arithmetic

In C++, pointer arithmetic is a powerful tool for memory management. "Pointer arithmetic refers to the operations that are valid to perform on pointers" (Sagar, 2024, p.1). Below is a list of valid pointer arithmetic operations:

1- **Incrementing and Decrementing Pointers**
When a pointer is incremented '*ptr++*', it points to the value of the derived data type which is in the next memory location. For instance, if '`ptr`' is an '`int*`', incrementing it moves it to the next integer in memory, which is typically 4 bytes ahead, as integers usually have a 4-byte size. Similarly, decrementing a pointer '*ptr--*' moves it to the previous element in memory.

```
int arr[3] = {10, 20, 30};
int* ptr = arr;

ptr++;  // Now points to arr[1] (20)
std::cout << *ptr << std::endl;  // Outputs 20

ptr--;  // Now points back to arr[0] (10)
std::cout << *ptr << std::endl;  // Outputs 10
```

2- **Addition of Constant to Pointers**
For example, if adding the constant '`2`' to a pointer, it will move forward the pointer by '`2 * size of int bytes`'

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

```
ptr = ptr + 2;  // Moves the pointer to arr[2] (30)
std::cout << *ptr << std::endl;  // Outputs 30
```

3- **Subtraction of Constant from Pointers**
Similar to adding a constant to a pointer, For example, if subtracting constant '`2`' to a pointer, it will move backward the pointer by '`2 * size of int bytes`'

```
int arr[5] = {10, 20, 30, 40, 50};
int* ptr = &arr[4];  // Points to arr[4] (50)

ptr = ptr - 3;  // Moves back 3 elements, now pointing to arr[1] (20)
std::cout << *ptr << std::endl;  // Outputs 20
```

4- **Subtraction of Two Pointers of the Same Type**
Subtracting of the same type will find the number of elements between them.
For example:

```
int arr[5] = {10, 20, 30, 40, 50};
int* ptr1 = &arr[1];  // Points to arr[1] (20)
int* ptr2 = &arr[4];  // Points to arr[4] (50)

int difference = ptr2 - ptr1;  // Difference in elements
std::cout << "Difference: " << difference << std::endl;  // Outputs 3
```

5- **Comparison of Pointers**
Tow pointers can be compared using relational operators (`>`, `<`, `>=`, `<=`, `==`, `!=`). This can tell if the two pointers are pointing to the same memory location or not.
For example:

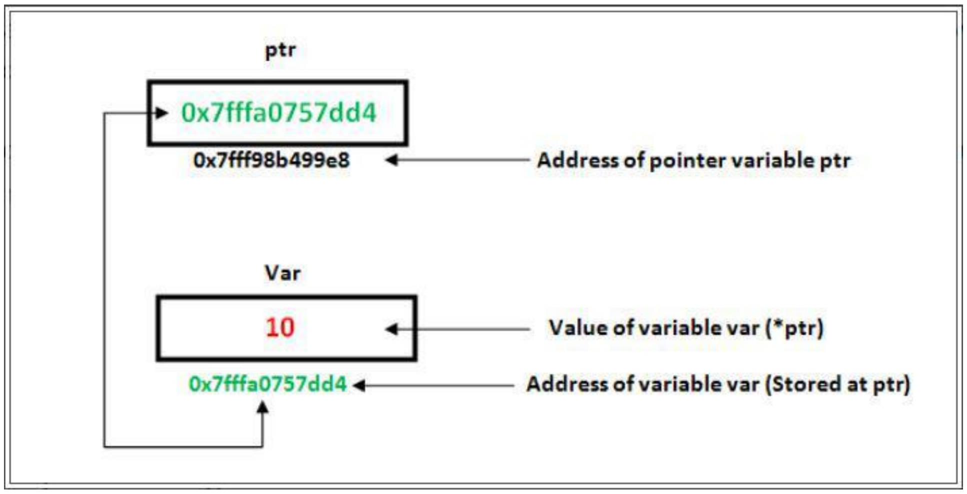
```
int num1 = 10, num2 = 20;
int* ptr1 = &num1;
int* ptr2 = &num2;

if (ptr1 != ptr2) {
    std::cout << "The pointers point to different locations." << std::endl;
} else {
    std::cout << "The pointers point to the same location." << std::endl;
}
// The pointers point to different locations.
```

Pointer arithmetic is powerful for manipulating and comparing pointers and the value they point to. This is possible through the use of the address-of and dereference operators (`*`) and (`&`) as shown in the prior example.

Address-of and Dereference Operators (*) and (&)

Understanding how pointers handle addresses and value stores in those addresses is crucial for coding without generating errors. At the center of the pointer's functionality are the address-of operator (`&`) and the indirection operator/Dereference operator (*). The diagram below shows the makeup of a pointer in C++.

Figure 1
Pointer


ptr
0x7fffa0757dd4
0x7fff98b499e8
Address of pointer variable ptr
Var
10
Value of variable var (*ptr)
0x7fffa0757dd4
Address of variable var (Stored at ptr)

Note: From "C++ Pointer Operators" by Raj (2022).

- The address-of operator (&) is a unary operator. It returns the memory address of the variable.
For example:

```
int x = 10;
int *ptr = &x;  // ptr now holds the address of x
```

- The dereference operator (*) is a unary operator. It returns the value of the variable present at the given address.
For example:

```
std::cout << *ptr;  // Outputs the value of x (10)
*ptr = 20;          // Modifies the value of x through the pointer
std::cout << x;     // Outputs the updated value of x (20)
```

The address-of and dereference operators allow developers to manage and manipulate memory addresses and the value stored in them through the pointer functionality. However, pointers can create significant safety risks if used improperly, they can cause undefined behavior, crashes, or security vulnerabilities such as buffer overflows. A good practice is to use smart pointers, called unique pointers in C++, rather than raw pointers, to ensure safer memory management.

Smart Pointers

Smart pointers in C++ are also referred to as unique pointers ('`unique_ptr`'). They automatically manage the dynamically allocated resources on the heap. They are just wrappers around regular raw pointers preventing errors such as forgetting to delete a pointer and causing a memory leak or accidentally deleting a pointer twice or in the wrong way (Pritamauddy, 2023). Additionally, a '`unique_ptr`' does not

share its raw pointer, meaning that it cannot be copied to another 'unique_ptr' (Whitney, 2021). Instead, it can only moved. This means that the ownership of the memory address of the values stored can be transferred from one 'unique_ptr' to another.
For example:

```
#include <iostream>
#include <memory>  // std::unique_ptr
#include <utility> // std::move

int main() {

    std::unique_ptr<int> ptr1 = std::make_unique<int>(10);

    std::cout << "Value pointed by ptr1: " << *ptr1 << std::endl;

    // Moving ownership from ptr1 to ptr2
    std::unique_ptr<int> ptr2 = std::move(ptr1);

    // ptr1 is now empty
    if (!ptr1) {
        std::cout << "ptr1 is now empty.<< std::endl;
    }

    // ptr2 now owns the address
    std::cout << "Value pointed by ptr2: " << *ptr2 << std::endl;

    // No need to delete manually; unique_ptr automatically deletes the resource when
    // it goes out of scope
    return 0;
}
```

As shown in the code example the 'unique_ptr' is safer to use than a raw pointer, it automatically manages access to the heap, protecting it from errors by wrapping around (managing) a raw pointer.

To summarize, pointers offer flexibility and precise control over memory, making them a powerful tool for applications where memory usage management and manipulation are needed. Pointers and arrays work well together; arrays are not necessarily required when using pointers and vice versa. Nonetheless, they are similar as both are derived data types and use direct memory access. However, they handle memory allocation and how elements are accessed differently. Pointers and the values they point to can be manipulated using pointer arithmetic and the operators (*) and (&). This comes with risks that can be mitigated by using the smart pointers ('unique_ptr'), which manage access to the heap, protecting it from errors.

-Alex

References:

Pritamauddy (2023, October 5). Unique_ptr in C++. GeeksforGeeks. https://www.geeksforgeeks.org/unique_ptr-in-cpp/

Raj (2022, December 20). *C++ Pointer operators*. GeeksforGeeks. https://www.geeksforgeeks.org/cpp-pointer-operators/

Rinkalktailor (2024, May 15). *Pointers vs array in C++*. GeeksforGeeks. https://www.geeksforgeeks.org/pointers-vs-array-in-cpp/

Sagar (2024, May 1). *C++ Pointer arithmetic*. GeeksforGeeks. https://www.geeksforgeeks.org/cpp-pointer-arithmetic/?ref=oin_asr26

Whitney, T., Junshen, T., Sharkey, K., Voss, J., Jones, M., Blome, M., Hogenson, G., & Cai, S. (2021, November 12). *How to: Create and use unique_ptr instances*. Microsoft Learn. https://learn.microsoft.com/en-us/cpp/cpp/how-to-create-and-use-unique-ptr-instances?view=msvc-170&viewFallbackFrom=vs-2019

Yashk. (2023, January 16). *C++ Program to access elements of an array using pointer*. GeeksforGeeks. https://www.geeksforgeeks.org/cpp-program-to-access-elements-of-an-array-using-pointer/