Pointers in C++: An Introduction

This is essential for understanding how we can create a dynamic (resizable) array. Python lists are dynamic arrays and understanding this lecture will help you understand how they are implemented.

We are not using pointers a lot, but I think it is well worth you understand some of the basic concepts.

- Pointers in C++ are variables that store memory addresses.
- They provide a powerful way to work with memory and data, especially when used in combination with structs.
- Pointers allow you to access and manipulate data in memory directly, which can be useful for tasks like dynamic memory allocation, creating data structures, and implementing efficient algorithms.

Let's explore why pointers are needed and how they work.

Motivation for Pointers:

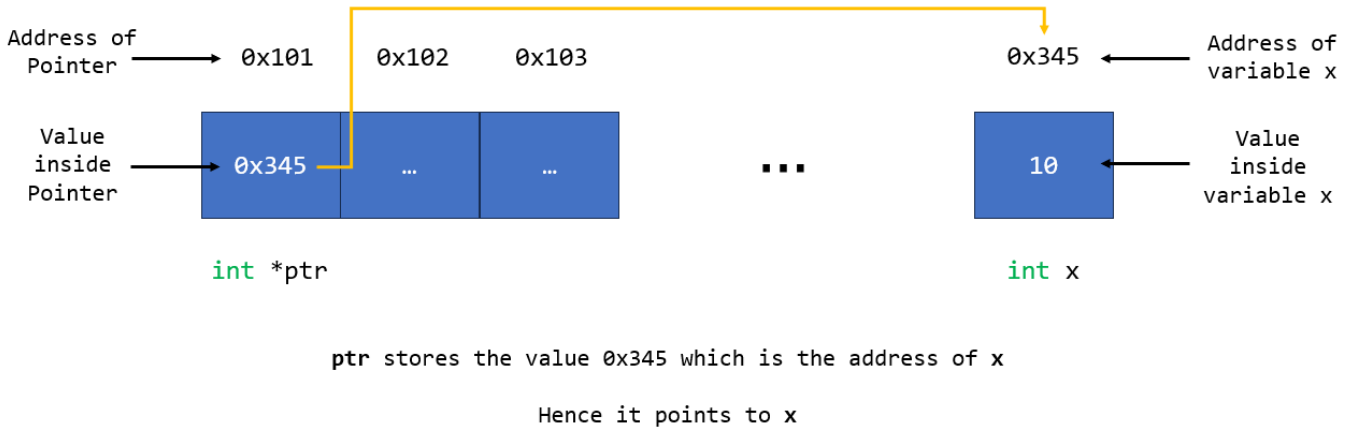
1. **Dynamic Memory Allocation**: Pointers are crucial for allocating and managing memory dynamically, allowing you to create and work with data structures of arbitrary sizes at runtime.

2. **Working with Data Structures**: Many data structures, such as linked lists and trees, rely on pointers to establish connections between elements and manage complex relationships.

3. **Efficiency**: Pointers provide direct access to memory, which can lead to more efficient data manipulation, especially in low-level programming.

How Pointers Work:

Pointers store memory addresses, allowing you to access the data located at those addresses.



The process involves two main operations: referencing and dereferencing.

- **Referencing**: To create a pointer, you use the address-of operator `&` to obtain the memory address of a variable.

- **Dereferencing**: To access the value stored at a memory address, you use the dereference operator `*`.

Examples

Now, let's see some examples:

Example 1: Pointer Basics

We declare a pointer of any type using the following format:

`dataType *ptrName = &var`

e.g. declare an `int` pointer and set to address of `x`

```
int x = 10;          // Create and integer variable
int *ptr = &x;       // Create a pointer and store the address of x
```

We can then do 3 things with our pointer.

- Get the memory address of our pointer - `&ptr`.
- Get the value of the `ptr` (memory address of `x`) - `ptr`
- Get the value stored in `x` - `*ptr`.

Here is an example.

```
#include <iostream>

int main() {
    int x = 42;    // Create an integer variable
    int *ptr = &x; // Create a pointer and store the address of x

    // Print the memory address of ptr
    std::cout << "Memory address of ptr: " << &ptr << std::endl;

    // Print the value of the pointer (address of x)
    std::cout << "Value of pointer (address of x): " << ptr << std::endl;

    // Dereference the pointer and print the value of x
    std::cout << "Dereferenced pointer (value of x): " << *ptr << std::endl;

    return 0;
}
```

Example 2: Basic Pointer Usage

In this example, we declare an integer `x`, a pointer `p` to an integer, and assign the address of `x` to the pointer. We can access the value of `x` using `*p`.

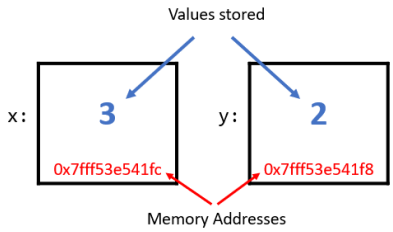
```
#include <iostream>

int main() {
    int x = 10;
    int* ptr;  // Declare a pointer to an integer
    ptr = &x;  // Assign the address of x to the pointer

    std::cout << "Value of x: " << x << "\n";
    std::cout << "Value through pointer ptr: " << *ptr << "\n";

    *ptr = 20;  // Modify the value through the pointer
    std::cout << "Value of x after modification: " << x << "\n";
    std::cout << "Value through pointer ptr: " << *ptr << "\n";

    return 0;
}
```

Example 3: Updating Variables with Pointers

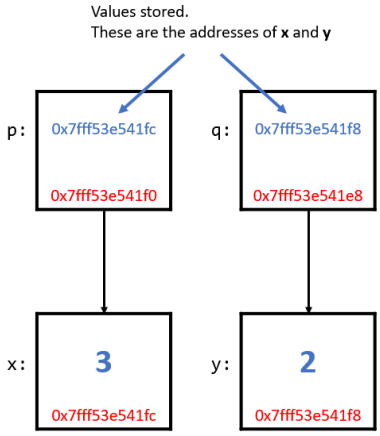
```
#include <iostream>

int main() {
    int x = 3;
    int y = 2;

    int* p = &x;     // Declare a pointer to x
    int* q = &y;     // Declare a pointer to y

    p = q;           // Point the pointer p to the address of q which points to y

    *p = 42;         // Overwrite the value stored at the address of p. This
overwrites y!

    /*
    p and q now both point to y.
    The value of x is still 3, but the value of y is 42
    */

    return 0;
}
```

Here is a visual explanation of what is happening.

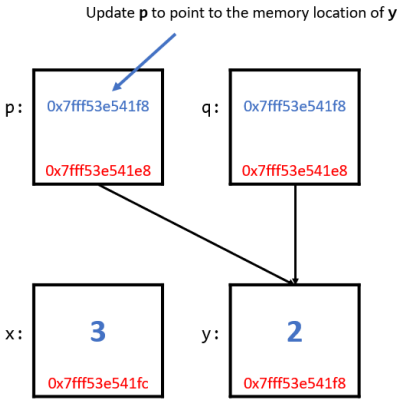
- Create the `int` variables `x` and `y`



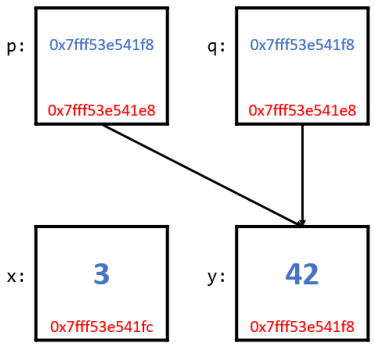
- Create the pointers `p` and `q` and point to `x` and `y` respectively.



- Update pointer `p` to point to the location of `y`.



- Update the value of `y` using the pointer `p`.



Example 4: Dynamic Memory Allocation

In this example, we dynamically allocate an integer array and populate it. Dynamic memory allocation with pointers allows us to create data structures of variable sizes at runtime.

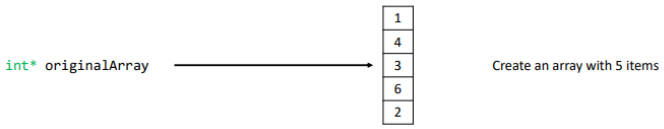
```
#include <iostream>

int main() {
    int* arr = new int[5];  // Dynamically allocate an array of 5 integers

    for (int i = 0; i < 5; i++) {
        arr[i] = i * 2;  // Initialize the array elements
    }

    std::cout << "Array elements: ";
    for (int i = 0; i < 5; i++) {
        std::cout << arr[i] << " ";
    }
    std::cout << "\n";

    delete[] arr;  // Deallocate the memory

    return 0;
}
```

Resizing an Array with Dynamic Memory Allocation

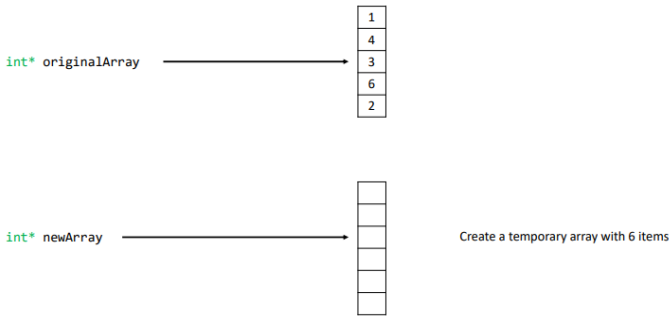
The final example gives us enough to start resizing an array. Rememeber an array is a fixed block of memory and we can't just add to the end of it because we will overwrite something else.

Therefore, we need to create a copy in memory if we want to add or delete to an array.

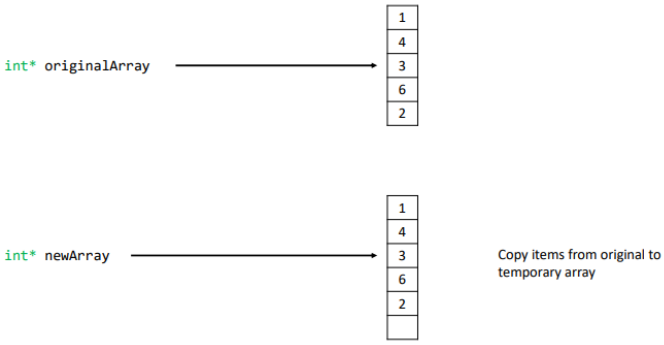
- Create a dynamic array of length `arraySize`. e.g. `5`. Store this as a pointer called `originalArray`



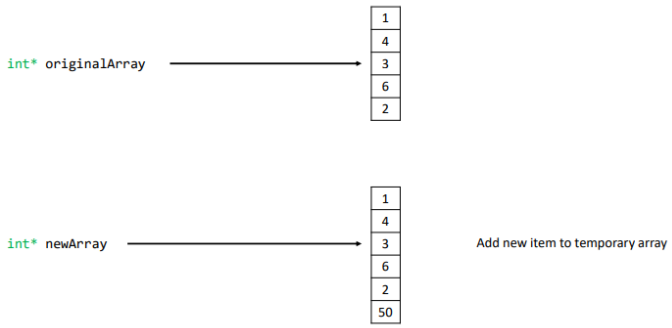
- Create a dynamic temporary array of size `arraySize + 1`. e.g. `6`. Store this as a pointer called `newArray`



- Copy items from `originalArray` to the `newArray`

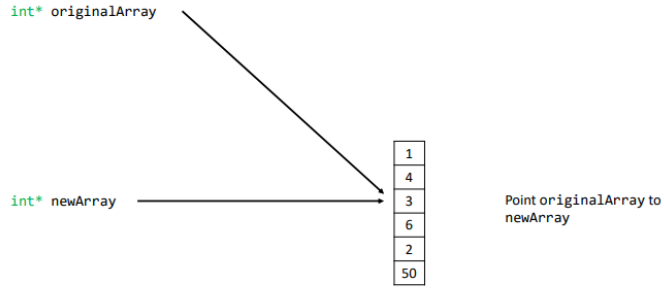


- Add the new item to the end of `newArray`

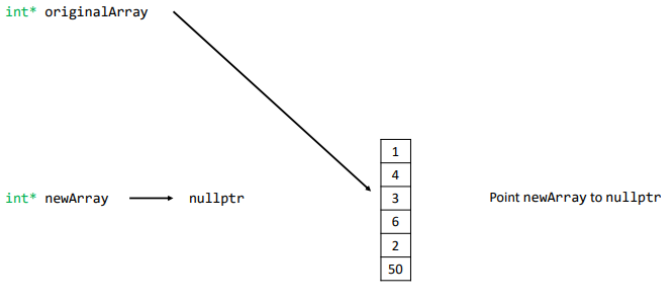


- Deallocate `originalArray`

- Update `originalArray` to point to `newArray`



- Point `newArray` to `nullptr`. Represents no value for a pointer.



You will find a C++ demonstration of this on the next page.

C++ Demonstration

```cpp
#include <iostream>

int main() {
    int* originalArray = new int[5]{1, 4, 3, 6, 2}; // Dynamically allocate an
array of 5 integers

    // Display the original array
    std::cout << "Original Array: ";
    for (int i = 0; i < 5; i++) {
        std::cout << originalArray[i] << " ";
    }
    std::cout << "\n";

    // Create a new array with a larger size (e.g., 6 elements)
    int newSize = 6;
    int* newArray = new int[newSize];

    // Copy elements from the original array to the new array
    for (int i = 0; i < 5; i++) {
        newArray[i] = originalArray[i];
    }

    // Add a new item at the end
    newArray[newSize - 1] = 50; // Adding 50 to the end

    // Deallocate the original array
    delete[] originalArray;

    // Update the original array pointer to point to the new array
    originalArray = newArray;

    // Display the updated array
    std::cout << "Updated Array: ";
    for (int i = 0; i < newSize; i++) {
        std::cout << originalArray[i] << " ";
    }
    std::cout << "\n";

    newArray = nullptr;

    return 0;
}
```

Summary of Pointers

- Pointers in C++ are variables that store memory addresses, providing a powerful way to work with memory and data.
- They are essential for dynamic memory allocation, creating data structures, and efficient data manipulation.
- Pointers store memory addresses, allowing direct access to the data located at those addresses.
- Key operations include referencing (using the `&` operator) to obtain the memory address of a variable and dereferencing (using the `*` operator) to access the value stored at a memory address.